F:\git\java\mar3\filemonitor\target\core-module\core-module-0.doc

0:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\cfg\NulsConfig.java
 */

```
package io.nuls.kernel.cfg;


import io.nuls.core.tools.cfg.IniEntity;

/**
 *
 * <p>
 * Used to manage system configuration items and system version information.
 *
 * @author: Niels Wang
 */
public class NulsConfig {

    /**
     * nuls
     * The version number of the underlying code for nuls.
     */
    public static String VERSION = "1.1.2";

    /**
     *
     * The encoding used by the nuls system.
     */
    public static String DEFAULT_ENCODING = "UTF-8";

    /**
     * nuls
     * The configuration items loaded in the nuls system configuration file.
     */
    public static IniEntity NULS_CONFIG;

    /**
     *
     * All the configuration items that are loaded in the module configuration file.
     */
```

```
    public static IniEntity MODULES_CONFIG;

}

1:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\constant\ErrorCode.java
 */

package io.nuls.kernel.constant;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.kernel.i18n.I18nUtils;

/**
 *
 * All of the system's return code management tools, all of the constants should be implemented
using this class.
 * This class integrates internationalization operations, and all information that needs
internationalization should be used.
 *
 * @author: Niels Wang
 */
public class ErrorCode {
    /**
     *
     * Internationalized encoding of message content.
     */
    private String msg;

    /**
     *
     * The return code is used to mark the unique result.
     */
    private String code;

    public ErrorCode() {

    }

    protected ErrorCode(String code) {
        this.code = code;
        this.msg = code;
```

```
        if (null == code) {
            throw new RuntimeException("the errorcode code cann't be null!");
        }
    }

    /**
     *
     * According to the system language Settings, return the string corresponding to the
internationalization encoding.
     *
     * @return String
     */
    public String getMsg() {
        return I18nUtils.get(msg);
    }

    /**
     *
     * return the English string corresponding to the internationalization encoding.
     *
     * @return String
     */
    @JsonIgnore
    public String getEnMsg() {
        return I18nUtils.getEn(msg);
    }

    public String getCode() {
        return code;
    }

    public static final ErrorCode init(String code) {
        return new ErrorCode(code);
    }

    @Override
    public boolean equals(Object obj) {
        if (null == obj) {
            return false;
        }
        if (!(obj instanceof ErrorCode)) {
            return false;
```

```
        }
        return code.equals(((ErrorCode) obj).getCode());
    }
}

2:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\constant\KernelErrorCode.java
 */
package io.nuls.kernel.constant;


/**
 *
 *
 * @author: Niels Wang
 */
public interface KernelErrorCode {

    ErrorCode SUCCESS = ErrorCode.init("10000");
    ErrorCode FAILED = ErrorCode.init("10001");
    ErrorCode SYS_UNKOWN_EXCEPTION = ErrorCode.init("10002");
    ErrorCode DATA_PARSE_ERROR = ErrorCode.init("10003");
    ErrorCode THREAD_REPETITION = ErrorCode.init("10004");
    ErrorCode LANGUAGE_CANNOT_SET_NULL = ErrorCode.init("10005");
    ErrorCode IO_ERROR = ErrorCode.init("10006");
    ErrorCode DATA_SIZE_ERROR = ErrorCode.init("10007");
    ErrorCode CONFIG_ERROR = ErrorCode.init("10008");
    ErrorCode SIGNATURE_ERROR = ErrorCode.init("10009");
    ErrorCode REQUEST_DENIED = ErrorCode.init("10010");
    ErrorCode DATA_SIZE_ERROR_EXTEND = ErrorCode.init("10011");
    ErrorCode PARAMETER_ERROR = ErrorCode.init("10012");
    ErrorCode NULL_PARAMETER = ErrorCode.init("10013");
    ErrorCode DATA_ERROR = ErrorCode.init("10014");
    ErrorCode DATA_NOT_FOUND = ErrorCode.init("10015");
    ErrorCode DOWNLOAD_VERSION_FAILD = ErrorCode.init("10016");
    ErrorCode PARSE_JSON_FAILD = ErrorCode.init("10017");
    ErrorCode FILE_OPERATION_FAILD = ErrorCode.init("10018");
    ErrorCode ILLEGAL_ACCESS_EXCEPTION = ErrorCode.init("10019");
    ErrorCode INSTANTIATION_EXCEPTION = ErrorCode.init("10020");
    ErrorCode UPGRADING = ErrorCode.init("10021");
    ErrorCode NOT_UPGRADING = ErrorCode.init("10022");
    ErrorCode VERSION_NOT_NEWEST = ErrorCode.init("10023");
```

```
    ErrorCode SERIALIZE_ERROR = ErrorCode.init("10024");
    ErrorCode DESERIALIZE_ERROR = ErrorCode.init("10025");
    ErrorCode HASH_ERROR = ErrorCode.init("10026");
    ErrorCode INSUFFICIENT_BALANCE = ErrorCode.init("10027");
    ErrorCode ADDRESS_IS_BLOCK_HOLE = ErrorCode.init("10028");
    ErrorCode ADDRESS_IS_NOT_BELONGS_TO_CHAIN = ErrorCode.init("10029");
    ErrorCode VALIDATORS_NOT_FULLY_EXECUTED = ErrorCode.init("10030");
    ErrorCode BLOCK_IS_NULL = ErrorCode.init("10031");
    ErrorCode VERSION_TOO_LOW = ErrorCode.init("10032");
    ErrorCode PUBKEY_REPEAT = ErrorCode.init("10033");
    ErrorCode COIN_OWNER_ERROR = ErrorCode.init("10034");
    ErrorCode NONEWVER = ErrorCode.init("10035");
}

3:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\constant\ModuleStatusEnum.java
 */
package io.nuls.kernel.constant;

/**
 *
 * The module runs state enumeration.
 *
 * @author: Niels Wang
 */
public enum ModuleStatusEnum {
    /**
     *
     * module not found
     */
    NOT_FOUND,

    /**
     *
     * uninitialized
     */
    UNINITIALIZED,

    /**
     *
     * initialized
     */
```

```
INITIALIZED,

/**
 *
 * initializing
 */
INITIALIZING,

/**
 *
 * starting
 */
STARTING,

/**
 *
 * running
 */
RUNNING,

/**
 *
 * stoped
 */
STOPED,

/**
 *
 * stopping
 */
STOPPING,

/**
 *
 * destoryed
 */
DESTROYED,

/**
 *
 * destorying
 */
```

```
    DESTROYING,

    /**
     *
     * Running exception
     */
    EXCEPTION,
    ;

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

4:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\constant\NulsConstant.java
 */
package io.nuls.kernel.constant;

import io.nuls.kernel.utils.AddressTool;

/**
 *
 * SYSTEM CONSTANT
 *
 * @author Niels
 */
public interface NulsConstant {

    /**
     * nuls
     * The nuls version upgrades the main configuration file
     */
    String NULS_VERSION_XML = "nuls-version.xml";

    byte[] BLACK_HOLE_ADDRESS =
AddressTool.getAddress("Nse5FeeiYk1opxdc5RqYpEWkiUDGNuLs");

    /**
     *
     * System configuration file name.
```

```
 */
String USER_CONFIG_FILE = "nuls.ini";

/**
 *
 * Module configuration file name.
 */
String MODULES_CONFIG_FILE = "modules.ini";

/**
 *
 * The name of the configuration item for the module startup class.
 */
String MODULE_BOOTSTRAP_KEY = "bootstrap";

/**
 * ----[ System] ----
 */
/**
 * section
 * The configuration item section name of the kernel module.
 */
String CFG_SYSTEM_SECTION = "System";

/**
 *
 * The field name of the language set in the system configuration.
 */
String CFG_SYSTEM_LANGUAGE = "language";

/**
 *
 * The field name of the code setting in the system configuration.
 */
String CFG_SYSTEM_DEFAULT_ENCODING = "encoding";

/**
 * id
 * The module id of micro kernel module
 */
short MODULE_ID_MICROKERNEL = 1;
```

```
    /**
     *
     * A consensus award for the type of trade.
     */
    int TX_TYPE_COINBASE = 1;

    /**
     *
     * the type of the transfer transaction
     */
    int TX_TYPE_TRANSFER = 2;
    /**
     *
     * Type of business data bearing transaction
     */
    int TX_TYPE_DATA = 10;

    /**
     *
     * Null placeholder.
     */
    byte[] PLACE_HOLDER = new byte[]{(byte) 0xFF, (byte) 0xFF, (byte) 0xFF, (byte) 0xFF};
    /**
     * 48
     * 48 bit integer data length.
     */
    int INT48_VALUE_LENGTH = 6;

    /**
     * utxo
     *
     *
     */
    long BlOCKHEIGHT_TIME_DIVIDE = 1000000000000L;

    /**
     *
     * Null placeholder.
     */
    byte[] SIGN_HOLDER = new byte[]{(byte) 0x00, (byte) 0x00};

}
```

5:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\constant\SeverityLevelEnum.java
```
 */
package io.nuls.kernel.constant;

/**
 *
 * <p>
 * The validator validates the out-of-date and needs to determine the severity of the error,
 * and the severity of the result is currently defined in the enumeration class.
 *
 * @author Niels
 */
public enum SeverityLevelEnum {

    /**
     *
     * Errors in the normal range, such as incomplete fields, incorrect time, etc., do not require penalties.
     */
    WRONG,

    /**
     *
     * A foul, but not a punishment.
     */
    NORMAL_FOUL,

    /**
     *
     * A flagrant foul must be punished.
     */
    FLAGRANT_FOUL;
}
```

6:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\constant\TransactionErrorCode.java
```
 *
 */

package io.nuls.kernel.constant;
```

```
/**
 * Created by ln on 2018/5/6.
 */
public interface TransactionErrorCode extends KernelErrorCode {

    ErrorCode UTXO_UNUSABLE = ErrorCode.init("31001");
    ErrorCode UTXO_STATUS_CHANGE = ErrorCode.init("31002");
    ErrorCode INVALID_INPUT = ErrorCode.init("31004");
    ErrorCode INVALID_AMOUNT = ErrorCode.init("31005");
    ErrorCode ORPHAN_TX = ErrorCode.init("31006");
    ErrorCode ORPHAN_BLOCK = ErrorCode.init("31007");
    ErrorCode TX_DATA_VALIDATION_ERROR = ErrorCode.init("31008");
    ErrorCode FEE_NOT_RIGHT = ErrorCode.init("31009");
    ErrorCode ROLLBACK_TRANSACTION_FAILED = ErrorCode.init("31010");
    ErrorCode TRANSACTION_REPEATED = ErrorCode.init("31011");
    ErrorCode TOO_SMALL_AMOUNT = ErrorCode.init("31012");
    ErrorCode TX_SIZE_TOO_BIG = ErrorCode.init("31013");
    ErrorCode SAVE_TX_ERROR = ErrorCode.init("31014");
    ErrorCode TX_NOT_EXIST = ErrorCode.init("31015");
    ErrorCode COINDATA_NOT_FOUND = ErrorCode.init("31016");
    ErrorCode TX_TYPE_ERROR = ErrorCode.init("31017");
    ErrorCode TX_NOT_EFFECTIVE = ErrorCode.init("31018");
}

7:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\constant\TxStatusEnum.java
 */
package io.nuls.kernel.constant;

/**
 *
 * Enumeration of transaction status
 *
 * @author Niels
 */
public enum TxStatusEnum {

    /**
     *
     * not packaged
     */
```

```
    UNCONFIRM,
    /**
     *
     * packaged and saved
     */
    CONFIRMED


}

8:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\context\NulsContext.java
 */
package io.nuls.kernel.context;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.model.Block;

import java.util.List;

/**
 *
 * System context provides core data sharing, service access and other functions.
 *
 * @author Niels
 */
public class NulsContext {

    /**
     * (1)
     * System protocol version.
     */
    public static volatile Integer CURRENT_PROTOCOL_VERSION = 1;

    /**
     *
     * contract address type
     */
    public static byte P2SH_ADDRESS_TYPE = 3;
```

```
    /**
     * 1
     */
    public static volatile Integer MAIN_NET_VERSION = 1;

    /**
     * HASH
     */
    public static Long CHANGE_HASH_SERIALIZE_HEIGHT;

    /**
     * idnuls,id“Ns”
     * The default chain id (nuls main chain), the chain id affects the generation of the address,
     * and the current address begins with "Ns".8964.
     */
    public static short DEFAULT_CHAIN_ID = 261;

    /**
     *
     * The default address type, a chain can contain several address types, and the address type is
contained in the address.
     */
    public static byte DEFAULT_ADDRESS_TYPE = 1;

    /**
     *
     * contract address type
     */
    public static byte CONTRACT_ADDRESS_TYPE = 2;


    /*
     *  chain name
     */
    public static String CHAIN_NAME = "NULS";

    public static String INITIAL_STATE_ROOT =
"56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421";

    /**
     * cache the best block
```

```
 */
private Block bestBlock;

/**
 * cache the genesis block
 */
private Block genesisBlock;

/**
 * 3hashblock
 */
public static byte REDPUNISH_BIFURCATION = 3;

/**
 *
 * Network latest block height.
 */
private Long netBestBlockHeight = 0L;

public int getStop() {
    return stop;
}

private int stop = 0;

public void exit(int stop) {
    this.stop = stop;
}

/**
 *
 */
public static volatile boolean mastUpGrade = false;

/**
 *
 * get the block height is 0
 *
 * @return block
 */
public Block getGenesisBlock() {
    while (genesisBlock == null) {
```

```
        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
        }
    }
    return genesisBlock;
}

public void setGenesisBlock(Block block) {
    this.genesisBlock = block;
}

public Block getBestBlock() {
    if (bestBlock == null) {
        bestBlock = getGenesisBlock();
    }
    return bestBlock;
}

/**
 *
 * Get the latest local height.
 *
 * @return long height
 */
public long getBestHeight() {
    if (bestBlock == null) {
        bestBlock = getGenesisBlock();
    }
    return bestBlock.getHeader().getHeight();
}

private NulsContext() {
}

private static final NulsContext NC = new NulsContext();

/**
 * get zhe only instance of NulsContext
 *
 * @return NulsContext
 */
```

```
    public static final NulsContext getInstance() {
        return NC;
    }


    public void setBestBlock(Block bestBlock) {
        if (bestBlock == null) {
            throw new RuntimeException("best block set to null!");
        }
        this.bestBlock = bestBlock;
//        Log.info("best height:"+bestBlock.getHeader().getHeight()+",
hash:"+bestBlock.getHeader().getHash());
    }

    /**
     *
     * Gets an instance based on the service type.
     *
     * @param tClass
     * @param <T>
     * @return
     */
    public static final <T> T getServiceBean(Class<T> tClass) {
        try {

            return SpringLiteContext.getBean(tClass);
        } catch (Exception e) {
            return getServiceBean(tClass, 0L);
        }
    }

    private static <T> T getServiceBean(Class<T> tClass, long l) {
        try {
            Thread.sleep(200L);
//            System.out.println("service"+tClass);
        } catch (InterruptedException e1) {
            Log.error(e1);
        }
        try {
            return SpringLiteContext.getBean(tClass);
        } catch (NulsRuntimeException e) {
            if (e.getCode().equals(KernelErrorCode.DATA_ERROR)) {
```

```
            throw e;
        }
        if (l > 1200) {
            Log.error(e);
            return null;
        }
        return getServiceBean(tClass, l + 10L);
    }
}

/**
 *
 * To get the latest height of the network, return to the local latest height
 * when the cached network's latest height is empty or less than the local height.
 *
 * @return long best height
 */
public Long getNetBestBlockHeight() {
    if (null != bestBlock && netBestBlockHeight < bestBlock.getHeader().getHeight()) {
        return bestBlock.getHeader().getHeight();
    }
    if (null == netBestBlockHeight) {
        return 0L;
    }
    return netBestBlockHeight;
}

/**
 *
 * Gets the latest height of the cached network.
 *
 * @return Long
 */
public Long getNetBestBlockHeightWithNull() {
    return netBestBlockHeight;
}

public void setNetBestBlockHeight(Long netBestBlockHeight) {
    this.netBestBlockHeight = netBestBlockHeight;
}

/**
```

```
     *
     * Gets all instances of this type according to the service type.
     *
     * @param tClass
     * @param <T>
     * @return
     */
    public static <T> List<T> getServiceBeanList(Class<T> tClass) {
        try {
            return SpringLiteContext.getBeanList(tClass);
        } catch (Exception e) {
            Log.error(e);
        }
        return null;
    }

}

9:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\exception\NulsException.java
 */
package io.nuls.kernel.exception;


import io.nuls.kernel.constant.ErrorCode;

/**
 * Created by Niels on 2017/10/9.
 */
public class NulsException extends Exception {

    private ErrorCode errorCode;
    private String code;
    private String message;

    /**
     * Constructs a new exception with the specified detail validator.  The
     * cause is not initialized, and may subsequently be initialized by
     * a call to {@link #initCause}.
     *
     * @param message the detail validator. The detail validator is saved for
     *                later retrieval by the {@link #getMessage()} method.
```

```
 */
public NulsException(ErrorCode message) {
    super(message.getMsg());
    this.errorCode = message;
    this.code = message.getCode();
    this.message = message.getMsg();
}


/**
 * Constructs a new exception with the specified detail validator and
 * cause.  Note that the detail validator associated with
 * {@code cause} is <i>not</i> automatically incorporated in
 * this exception's detail validator.
 *
 * @param message the detail validator (which is saved for later retrieval
 *                by the {@link #getMessage()} method).
 * @param cause   the cause (which is saved for later retrieval by the
 *                {@link #getCause()} method).  (A <tt>null</tt> value is
 *                permitted, and indicates that the cause is nonexistent or
 *                unknown.)
 * @since 1.4
 */
public NulsException(ErrorCode message, Throwable cause) {
    super(cause);
    this.errorCode = message;
    this.code = message.getCode();
    this.message = message.getMsg();
}

/**
 * Constructs a new exception with the specified cause and a detail
 * validator of <tt>(cause==null ? null : cause.toString())</tt> (which
 * typically contains the class and detail validator of <tt>cause</tt>).
 * This constructor is useful for exceptions that are little more than
 * wrappers for other throwables (for example, {@link
 * java.security.PrivilegedActionException}).
 *
 * @param cause the cause (which is saved for later retrieval by the
 *              {@link #getCause()} method).  (A <tt>null</tt> value is
 *              permitted, and indicates that the cause is nonexistent or
 *              unknown.)
```

```
     * @since 1.4
     */
    public NulsException(Throwable cause) {
        super(cause);
    }

    /**
     * Constructs a new exception with the specified detail validator,
     * cause, suppression enabled or disabled, and writable stack
     * trace enabled or disabled.
     *
     * @param message           the detail validator.
     * @param cause             the cause.  (A {@code null} value is permitted,
     *                          and indicates that the cause is nonexistent or unknown.)
     * @param enableSuppression  whether or not suppression is enabled
     *                          or disabled
     * @param writableStackTrace whether or not the stack trace should
     *                          be writable
     * @since 1.7
     */
    protected NulsException(ErrorCode message, Throwable cause,
                            boolean enableSuppression,
                            boolean writableStackTrace) {
        super(message.getMsg(), cause, enableSuppression, writableStackTrace);
        this.errorCode = message;
        this.code = message.getCode();
        this.message = message.getMsg();
    }

    public ErrorCode getErrorCode() {
        return errorCode;
    }
}

10:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\exception\NulsRuntimeException.java
 */
package io.nuls.kernel.exception;


import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.constant.ErrorCode;
```

```
/**
 * @author Niels
 */
public class NulsRuntimeException extends RuntimeException {

    private String code;
    private String message;
    private ErrorCode errorCode;

    /**
     * Constructs a new exception with the specified detail validator.  The
     * cause is not initialized, and may subsequently be initialized by
     * a call to {@link #initCause}.
     *
     * @param message the detail validator. The detail validator is saved for
     *                later retrieval by the {@link #getMessage()} method.
     */
    public NulsRuntimeException(ErrorCode message) {
        super(message.getMsg());
        this.code = message.getCode();
        this.message = message.getMsg();
        this.errorCode = message;
    }

    /**
     * Constructs a new exception with the specified detail validator and
     * cause.  Note that the detail validator associated with
     * {@code cause} is <i>not</i> automatically incorporated in
     * this exception's detail validator.
     *
     * @param message the detail validator (which is saved for later retrieval
     *                by the {@link #getMessage()} method).
     * @param cause   the cause (which is saved for later retrieval by the
     *                {@link #getCause()} method).  (A <tt>null</tt> value is
     *                permitted, and indicates that the cause is nonexistent or
     *                unknown.)
     * @since 1.4
     */
    public NulsRuntimeException(ErrorCode message, Throwable cause) {
        super(message.getMsg(), cause);
        this.code = message.getCode();
```

```
    this.message = message.getMsg();
    this.errorCode = message;
    ;
}

/**
 * Constructs a new exception with the specified cause and a detail
 * validator of <tt>(cause==null ? null : cause.toString())</tt> (which
 * typically contains the class and detail validator of <tt>cause</tt>).
 * This constructor is useful for exceptions that are little more than
 * wrappers for other throwables (for example, {@link
 * java.security.PrivilegedActionException}).
 *
 * @param cause the cause (which is saved for later retrieval by the
 *              {@link #getCause()} method).  (A <tt>null</tt> value is
 *              permitted, and indicates that the cause is nonexistent or
 *              unknown.)
 * @since 1.4
 */
public NulsRuntimeException(Throwable cause) {
    super(cause);
}

/**
 * Constructs a new exception with the specified detail validator,
 * cause, suppression enabled or disabled, and writable stack
 * trace enabled or disabled.
 *
 * @param message            the detail validator.
 * @param cause              the cause.  (A {@code null} value is permitted,
 *                      and indicates that the cause is nonexistent or unknown.)
 * @param enableSuppression  whether or not suppression is enabled
 *                      or disabled
 * @param writableStackTrace whether or not the stack trace should
 *                      be writable
 * @since 1.7
 */
protected NulsRuntimeException(ErrorCode message, Throwable cause,
                          boolean enableSuppression,
                          boolean writableStackTrace) {
    super(message.getMsg(), cause, enableSuppression, writableStackTrace);
    this.code = message.getCode();
```

```
        this.message = message.getMsg();
        this.errorCode = message;
    }

//    public NulsRuntimeException(ErrorCode errorCode, String msg) {
//        super(msg);
//        this.code = errorCode.getCode();
//        this.message = errorCode.getMsg() + ":" + msg;
//        this.errorCode = errorCode;
//    }

    @Override
    public String getMessage() {
        if (StringUtils.isBlank(message)) {
            return super.getMessage();
        }
        return message;
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public ErrorCode getErrorCode() {
        return errorCode;
    }
}

11:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\exception\NulsVerificationException.java
 */
package io.nuls.kernel.exception;

import io.nuls.kernel.constant.ErrorCode;

/**
 * Created by facjas on 2017/10/31.
 */
```

```
public class NulsVerificationException extends NulsRuntimeException {

   /* public NulsVerificationException(String msg) {
        super(KernelErrorCode.VERIFICATION_FAILD, msg);
    }*/

    public NulsVerificationException(ErrorCode errorCode) {
        super(errorCode);
    }

    public NulsVerificationException(ErrorCode errorCode, Throwable e) {
        super(errorCode, e);
    }

//    public NulsVerificationException(ErrorCode errorCode, String msg) {
//        super(errorCode, msg);
//    }
}

12:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\func\TimeService.java
 */

package io.nuls.kernel.func;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.thread.manager.TaskManager;
import org.apache.commons.net.ntp.NTPUDPClient;
import org.apache.commons.net.ntp.TimeInfo;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * Time service class:Used to synchronize network standard time.
 *
 * @author vivi
 */
public class TimeService implements Runnable {
```

```
private TimeService() {
    urlList.add("sgp.ntp.org.cn");
    urlList.add("cn.ntp.org.cn");
    urlList.add("time1.apple.com");
    urlList.add("ntp3.aliyun.com");
    urlList.add("ntp5.aliyun.com");
    urlList.add("us.ntp.org.cn");
    urlList.add("kr.ntp.org.cn");
    urlList.add("de.ntp.org.cn");
    urlList.add("jp.ntp.org.cn");
    urlList.add("ntp7.aliyun.com");
}

private static TimeService instance = new TimeService();

public static TimeService getInstance() {
    return instance;
}

/**
 * url
 */
private List<String> urlList = new ArrayList<>();

/**
 *
 * Time migration gap trigger point, which can cause local time reset, unit milliseconds.
 **/
public static final long TIME_OFFSET_BOUNDARY = 3000L;

/**
 *
 * Resynchronize the interval.
 */
private static final long NET_REFRESH_TIME = 10 * 60 * 1000L;   // 10 minutes;

/**
 *
 */
private static long netTimeOffset;

/**
```

```
 *
 * The last synchronization point.
 */
private static long lastSyncTime;

/**
 *
 */
private void syncWebTime() {

    int count = 0;
    long sum = 0L;

    for (int i = 0; i < urlList.size(); i++) {
        long localBeforeTime = System.currentTimeMillis();

        long netTime = getWebTime(urlList.get(i));

        if (netTime == 0) {
            continue;
        }

        long localEndTime = System.currentTimeMillis();

        long value = (netTime + (localEndTime - localBeforeTime) / 2) - localEndTime;
        count++;
        sum += value;
    }
    if (count > 0) {
        netTimeOffset = sum / count;
    }

    lastSyncTime = currentTimeMillis();
}

/**
 *
 * todo
 *
 * @return long
 */
private long getWebTime(String address) {
```

```
    try {
        NTPUDPClient client = new NTPUDPClient();
        client.open();
        client.setDefaultTimeout(1000);
        client.setSoTimeout(1000);
        InetAddress inetAddress = InetAddress.getByName(address);
        Log.debug("start ask time....");
        TimeInfo timeInfo = client.getTime(inetAddress);
        Log.debug("done!");
        return timeInfo.getMessage().getTransmitTimeStamp().getTime();
    } catch (Exception e) {
        return 0L;
    }
}

/**
 *
 * Start the time synchronization thread.
 */
public void start() {
    Log.debug("----------- TimeService start ----------");
    TaskManager.createAndRunThread((short) 1, "TimeService", this, true);
}

/**
 *
 * Loop call synchronous network time method.
 */
@Override
public void run() {
    long lastTime = System.currentTimeMillis();
    syncWebTime();
    while (true) {
        long newTime = System.currentTimeMillis();

        if (Math.abs(newTime - lastTime) > TIME_OFFSET_BOUNDARY) {
            Log.debug("local time changed {}", newTime - lastTime);
            syncWebTime();

        } else if (currentTimeMillis() - lastSyncTime > NET_REFRESH_TIME) {
            //
            syncWebTime();
```

```
            }
            lastTime = newTime;
            try {
                Thread.sleep(500L);
            } catch (InterruptedException e) {

            }
        }
    }


    /**
     *
     * Gets the current network time in milliseconds.
     *
     * @return long
     */
    public static long currentTimeMillis() {
        return System.currentTimeMillis() + netTimeOffset;
    }

    /**
     *
     * Gets the network time offset.
     *
     * @return long
     */
    public static long getNetTimeOffset() {
        return netTimeOffset;
    }
}

13:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\i18n\I18nUtils.java
 */
package io.nuls.kernel.i18n;

import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
```

```
import java.io.*;
import java.net.URL;
import java.net.URLDecoder;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

/**
 *
 * Internationalized tools can convert information encoding into
 * a readable string of different languages, depending on the language set by the system.
 *
 * @author Niels
 */
public class I18nUtils {
    /**
     *
     * The language pool contains all the configured language package data.
     */
    private static final Map<String, Properties> ALL_MAPPING = new HashMap<>();

    /**
     *
     * The system currently selects the language package.
     */
    private static Properties nowMapping = new Properties();

    /**
     *
     * default language is English
     */
    private static String key = "en";

    /**
     *
     * default properties file folder
     */
    private static final String FOLDER = "languages";

    /**
     *
```

```
     * Load all the language packages before the first call.
     */
    static {
        //load all language properties
        try {
            URL furl = I18nUtils.class.getClassLoader().getResource(FOLDER);
            if (null != furl) {
                File folderFile = new File(URLDecoder.decode(furl.getPath(),"UTF-8"));
                for (File file : folderFile.listFiles()) {
                    InputStream is = new FileInputStream(file);
                    Properties prop = new Properties();
                    prop.load(new InputStreamReader(is, NulsConfig.DEFAULT_ENCODING));
                    String key = file.getName().replace(".properties", "");
                    ALL_MAPPING.put(key, prop);
                }
            }
        } catch (IOException e) {
            Log.error(e);
        }
    }

    //    /**
//     *
//     * Set up the system language and switch the language package.
//     *
//     * @param lang /Language identification
//     */
    public static void setLanguage(String lang) throws NulsException {
        if (StringUtils.isBlank(lang)) {
            throw new NulsException(KernelErrorCode.LANGUAGE_CANNOT_SET_NULL);
        }
        key = lang;
        nowMapping = ALL_MAPPING.get(lang);
    }

    /**
     *
     * Obtain a translated message body based on the information encoding.
     *
     * @param id
     * @return String /The translated string.
     */
```

```
    public static String get(String id) {
        if (nowMapping == null) {
            nowMapping = ALL_MAPPING.get(key);
        }
        return nowMapping.getProperty(id + "");
    }

    /**
     *
     * Obtain a translated English message body based on the information encoding.
     *
     * @param id
     * @return String /The translated English string.
     */
    public static String getEn(String id) {
        return ALL_MAPPING.get("en").getProperty(id + "");
    }

    /**
     *
     * Determines whether a language package has been loaded.
     *
     * @param lang /Language identification
     * @return /Determine the results
     */
    public static boolean hasLanguage(String lang) {
        return ALL_MAPPING.containsKey(lang);
    }

    /**
     *
     * Gets the language id that the current system has set.
     *
     * @return String /Language identification
     */
    public static String getLanguage() {
        return key;
    }
}
```

14:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\annotation\Autowired.java

```
 */
package io.nuls.kernel.lite.annotation;

import java.lang.annotation.*;

/**
 * bean
 * After the annotation is marked with the attributes used to mark the bean,
 * the system is automatically assigned to the field during the initialization phase.
 *
 * @author Niels Wang
 */
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {

    /**
     * bean
     * Depending on the bean name, it can be empty and the default is empty.
     *
     * @return /bean name
     */
    String value() default "";
}

15:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\annotation\Cmd.java
 */
package io.nuls.kernel.lite.annotation;

import java.lang.annotation.*;

/**
 *
 *
 * This annotation is used to identify the type, and when the annotation is marked on the type,
 * the system instantiates the object in the initialization phase and assigns the attributes that need
to be automatically loaded.
 * The type generated by the annotation identifier is not instantiated using a dynamic proxy and is
not intercepted by the default interceptor.
 *
```

```
 * @author: Niels Wang
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Cmd {

    /**
     * bean
     * the bean name, it can be empty and the default is empty.
     *
     * @return /bean name
     */
    String value() default "";
}

16:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\annotation\Component.java
 */
package io.nuls.kernel.lite.annotation;

import java.lang.annotation.*;

/**
 *
 *
 * This annotation is used to identify the type, and when the annotation is marked on the type,
 * the system instantiates the object in the initialization phase and assigns the attributes that need to be automatically loaded.
 * The type generated by the annotation identifier is not instantiated using a dynamic proxy and is not intercepted by the default interceptor.
 *
 * @author: Niels Wang
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Component {

    /**
     * bean
     * the bean name, it can be empty and the default is empty.
```

```
     *
     * @return /bean name
     */
    String value() default "";
}

17:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\annotation\Interceptor.java
 */
package io.nuls.kernel.lite.annotation;

import io.nuls.kernel.lite.core.interceptor.BeanMethodInterceptor;

import java.lang.annotation.*;

/**
 * {@link BeanMethodInterceptor}
 * The interceptor annotation, annotated with the object of the annotation, needs to implement the {@link BeanMethodInterceptor} interface,
 * which intercepts all methods or objects that annotate the specified annotation.
 *
 * @author: Niels Wang
 */

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Interceptor {
    /**
     * ,,
     * The interceptor CARES about the type of annotation, the annotations can be marked on the type or method, on the type,
     * can intercept all the methods of the class, on the way, only intercept marked method, cannot be empty
     *
     * @return Class
     */
    Class value();
}

18:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\annotation\Service.java
```

```
 */
package io.nuls.kernel.lite.annotation;

import java.lang.annotation.*;

/**
 *
 *
 * This annotation is used to identify the type, and when the annotation is marked on the type,
 * the system instantiates the object in the initialization phase and assigns the attributes that need to be automatically loaded.
 * The type generated by the annotation identifier is instantiated using a dynamic proxy and is intercepted by the default interceptor.
 *
 * @author: Niels Wang
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Service {

    /**
     * bean
     * the bean name, it can be empty and the default is empty.
     *
     * @return /bean name
     */
    String value() default "";
}

19:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\core\bean\InitializingBean.java
 */

package io.nuls.kernel.lite.core.bean;

import io.nuls.kernel.exception.NulsException;

/**
 *
 * Initialize the object interface.
 *
```

```
 * @author: Niels Wang
 */
public interface InitializingBean {

    //    /**
//     *
//     * This method is invoked after all properties are set, and is used to assist object initialization.
//     * @throws NulsException
//     */
    void afterPropertiesSet() throws NulsException;

}

20:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\DefaultMethodInterceptor.java
 */
package io.nuls.kernel.lite.core;

import io.nuls.kernel.lite.core.interceptor.BeanMethodInterceptorManager;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

/**
 * aop
 * The system's default method interceptor is used for the aop underlying implementation.
 *
 * @author: Niels Wang
 */
class DefaultMethodInterceptor implements MethodInterceptor {
    /**
     *
     * Intercept method
     *
     * @param obj         /Method owner
     * @param method      /Method definition
     * @param params      /Method parameter list
     * @param methodProxy
     * @return /Returns the return value of the intercepting method, which can be processed and
replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
```

```
     */
    @Override
    public Object intercept(Object obj, Method method, Object[] params, MethodProxy
methodProxy) throws Throwable {
        if (null == method.getDeclaredAnnotations() || method.getDeclaredAnnotations().length == 0)
{
            return methodProxy.invokeSuper(obj, params);
        }
        return BeanMethodInterceptorManager.doInterceptor(method.getDeclaredAnnotations(), obj,
method, params, methodProxy);
    }
}

21:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\interceptor\BeanMethodInterceptor.java
 */
package io.nuls.kernel.lite.core.interceptor;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

/**
 *
 * The interceptor interface used in the system object manager, when you want to intercept some
methods,
 * you need to define your own interceptor to implement the interface.
 *
 * @author: Niels Wang
 */
public interface BeanMethodInterceptor {
    /**
     *
     * When an interceptor intercepts a method, the method is used to logically determine whether
the intercepting method is called in the method,
     * and it can do some business operations before and after the call.
     *
     * @param annotation       /Annotation instances of the intercepting method.
     * @param object           /Method owner
     * @param method           /Method definition
     * @param params           /Method parameter list
     * @param interceptorChain
     * @return /Returns the return value of the intercepting method, which can be processed and
```

replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    Object intercept(Annotation annotation, Object object, Method method, Object[] params,
BeanMethodInterceptorChain interceptorChain) throws Throwable;
}

22:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\interceptor\BeanMethodInterceptorChain.java
 */
package io.nuls.kernel.lite.core.interceptor;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * Method the interceptor chain: one method can be more interceptors to intercept,
 * between multiple interceptors sequence formed a chain of interceptors, behind each blocker can
decide whether to continue the interceptor
 *
 * @author Niels Wang
 */
public class BeanMethodInterceptorChain {

    //   /**
//   *
//   * List of interceptors in the interceptors chain.
//   */
    protected List<BeanMethodInterceptor> interceptorList = new ArrayList<>();
    //
//   /**
//   *
//   * Thread-safe execution cache to mark the current execution progress.
//   */
    private ThreadLocal<Integer> index = new ThreadLocal<>();

    //   /**

```
//     *
//     * Method agent cache, thread safe.
//     */
    private ThreadLocal<MethodProxy> methodProxyThreadLocal = new ThreadLocal<>();

    //    /**
//     *
//     * Add a method interceptor to the chain.
//     *
//     * @param interceptor
//     */
    protected void add(BeanMethodInterceptor interceptor) {
        interceptorList.add(interceptor);
    }

    /**
     *
     * Puts a method in the interceptor chain to retrieve the returned result.
     *
     * @param annotation  /Annotation instances of the intercepting method.
     * @param object      /Method owner
     * @param method      /Method definition
     * @param params      /Method parameter list
     * @param methodProxy
     * @return /Returns the return value of the intercepting method, which can be processed and replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    public Object startInterceptor(Annotation annotation, Object object, Method method, Object[] params, MethodProxy methodProxy) throws Throwable {
        methodProxyThreadLocal.set(methodProxy);
        index.set(-1);
        Object result = null;
        try {
            result = execute(annotation, object, method, params);
        } finally {
            index.remove();
            methodProxyThreadLocal.remove();
        }
        return result;
    }
```

```
    //
//    /**
//     *
//     * Call a specific interceptor.
//     */
    public Object execute(Annotation annotation, Object object, Method method, Object[] params)
throws Throwable {
        index.set(1 + index.get());
        if (index.get() == interceptorList.size()) {
            return methodProxyThreadLocal.get().invokeSuper(object, params);
        }
        BeanMethodInterceptor interceptor = interceptorList.get(index.get());
        return interceptor.intercept(annotation, object, method, params, this);
    }

}
```

23:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\lite\core\interceptor\BeanMethodInterceptorManager.java

```
 */
package io.nuls.kernel.lite.core.interceptor;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 *
 * Interceptor manager.
 *
 * @author Niels Wang
 */
public class BeanMethodInterceptorManager {

    /**
     *
```

```
     * The interceptor pool
     */
    private static final Map<Class, BeanMethodInterceptorChain> INTERCEPTOR_MAP = new
HashMap<>();

    /**
     *
     * Add a method interceptor to the manager.
     *
     * @param annotationType
     * @param interceptor
     */
    public static void addBeanMethodInterceptor(Class annotationType, BeanMethodInterceptor
interceptor) {
        BeanMethodInterceptorChain interceptorChain = INTERCEPTOR_MAP.get(annotationType);
        if (null == interceptorChain) {
            interceptorChain = new BeanMethodInterceptorChain();
        }
        interceptorChain.add(interceptor);
        INTERCEPTOR_MAP.put(annotationType, interceptorChain);
    }

    /**
     *
     * Implement a method that assembles the interceptor chain according to the method's
annotations and puts it into the interceptor chain.
     *
     * @param annotations /Method annotated list of annotations.
     * @param object      /Method owner
     * @param method      /Method definition
     * @param params      /Method parameter list
     * @param methodProxy
     * @return /Returns the return value of the intercepting method, which can be processed and
replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    public static Object doInterceptor(Annotation[] annotations, Object object, Method method,
Object[] params, MethodProxy methodProxy) throws Throwable {
        List<Annotation> annotationList = new ArrayList<>();
        List<BeanMethodInterceptorChain> chainList = new ArrayList<>();
        for (Annotation ann : annotations) {
            BeanMethodInterceptorChain chain = INTERCEPTOR_MAP.get(ann.annotationType());
```

```
            if (null != chain) {
                chainList.add(chain);
                annotationList.add(ann);
            }
        }
        if (annotationList.isEmpty()) {
            return methodProxy.invokeSuper(object, params);
        }
        MultipleBeanMethodInterceptorChain chain = new
MultipleBeanMethodInterceptorChain(annotationList, chainList);
        return chain.startInterceptor(null, object, method, params, methodProxy);
    }
}

24:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\interceptor\MultipleBeanMethodInterceptorCha
in.java
 */
package io.nuls.kernel.lite.core.interceptor;

import io.nuls.core.tools.log.Log;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

/**
 * :
 * Multiple interceptors chain.Only when one method has multiple connector chains,
 * The chain is initialized and assembled every time a method is executed.
 *
 * @author Niels Wang
 */
public class MultipleBeanMethodInterceptorChain extends BeanMethodInterceptorChain {

    /**
     *
     */
    protected List<Annotation> annotationList = new ArrayList<>();
```

```
    /**
     *
     * Progress mark
     */
    protected Integer index = -1;

    /**
     *
     * Method proxy object
     */
    protected MethodProxy methodProxy;


    //    /**
//     *
//     * Initialize multiple interceptor chains.
//     *
//     * @param annotations
//     * @param chainList
//     */
    public MultipleBeanMethodInterceptorChain(List<Annotation> annotations,
List<BeanMethodInterceptorChain> chainList) {
        if (null == annotations || annotations.isEmpty()) {
            return;
        }
        for (int i = 0; i < annotations.size(); i++) {
            fillInterceptorList(annotations.get(i), chainList.get(i));
        }
    }

    //    /**
//     *
//     * Add an interceptor chain to the multiple interceptor chain,
//     *
//     * @param annotation                The comment for the interceptor chain.
//     * @param beanMethodInterceptorChain
//     */
    private void fillInterceptorList(Annotation annotation, BeanMethodInterceptorChain
beanMethodInterceptorChain) {
        for (BeanMethodInterceptor interceptor : beanMethodInterceptorChain.interceptorList) {
            annotationList.add(annotation);
            interceptorList.add(interceptor);
```

```
        }
    }

    //
//    /**
//     *
//     * Start executing the interceptor chain.
//     *
//     * @param annotation  /Annotation instances of the intercepting method.
//     * @param object      /Method owner
//     * @param method      /Method definition
//     * @param params      /Method parameter list
//     * @param methodProxy
//     */
    @Override
    public Object startInterceptor(Annotation annotation, Object object, Method method, Object[]
params, MethodProxy methodProxy) throws Throwable {
        this.methodProxy = methodProxy;
        index = -1;
        Object result = null;
        try {
            result = execute(null, object, method, params);
        } catch (Exception e) {
            Log.error(e);
            throw e;
        } finally {
            index = -1;
            this.methodProxy = null;
        }
        return result;
    }

    /**
     *
     * Call a specific interceptor.
     *
     * @param annotation /Annotation instances of the intercepting method.
     * @param object     /Method owner
     * @param method     /Method definition
     * @param params     /Method parameter list
     * @return /Returns the return value of the intercepting method, which can be processed and
replaced.
```

```
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    @Override
    public Object execute(Annotation annotation, Object object, Method method, Object[] params)
throws Throwable {
        index += 1;
        if (index == interceptorList.size()) {
            return methodProxy.invokeSuper(object, params);
        }
        annotation = annotationList.get(index);
        BeanMethodInterceptor interceptor = interceptorList.get(index);
        return interceptor.intercept(annotation, object, method, params, this);
    }
}

25:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\ModularServiceMethodInterceptor.java
 */

package io.nuls.kernel.lite.core;

import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.lite.core.interceptor.BeanMethodInterceptorManager;
import io.nuls.kernel.lite.exception.BeanStatusException;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.module.manager.ServiceManager;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 *
 * System default service interceptor.
 *
```

```
 * @author Niels
 */
public class ModularServiceMethodInterceptor implements MethodInterceptor {
    /**
     *
     * Thread-safe interceptors perform progress identification.
     */
    private ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

    /**
     *
     * Intercept method
     *
     * @param obj         /Method owner
     * @param method      /Method definition
     * @param params      /Method parameter list
     * @param methodProxy
     * @return /Returns the return value of the intercepting method, which can be processed and
replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    @Override
    public Object intercept(Object obj, Method method, Object[] params, MethodProxy
methodProxy) throws Throwable {
//        Log.debug(method.toString());
        threadLocal.set(0);
        Throwable throwable = null;
        while (threadLocal.get() < 100) {
            try {
                return this.doIntercept(obj, method, params, methodProxy);
            } catch (BeanStatusException e) {
                threadLocal.set(threadLocal.get() + 1);
                throwable = e;
                Thread.sleep(200L);
            }
        }
        throw throwable;
    }

    /**
     *
     * The actual intercept method
```

```
     *
     * @param obj         /Method owner
     * @param method      /Method definition
     * @param params      /Method parameter list
     * @param methodProxy
     * @return /Returns the return value of the intercepting method, which can be processed and
replaced.
     * @throws Throwable /This method may throw an exception, handle with care.
     */
    private Object doIntercept(Object obj, Method method, Object[] params, MethodProxy
methodProxy) throws Throwable {
        List<Annotation> annotationList = new ArrayList<>();
        if (!method.getDeclaringClass().equals(Object.class)) {
            String className = obj.getClass().getCanonicalName();
            className = className.substring(0, className.indexOf("$$"));
            Class clazz = Class.forName(className);
            fillAnnotationList(annotationList, clazz, method);
            BaseModuleBootstrap module = ServiceManager.getInstance().getModule(clazz);
            if (module == null) {
                throw new BeanStatusException(KernelErrorCode.DATA_ERROR);
            }
            if (module.getModuleId() != NulsConstant.MODULE_ID_MICROKERNEL &&
                    module.getStatus() != ModuleStatusEnum.STARTING &&
                    module.getStatus() != ModuleStatusEnum.RUNNING) {
                throw new BeanStatusException(KernelErrorCode.DATA_ERROR);
            }
            boolean isOk = SpringLiteContext.checkBeanOk(obj);
            if (!isOk) {
                throw new BeanStatusException(KernelErrorCode.DATA_ERROR);
            }
        }
        if (annotationList.isEmpty()) {
            return methodProxy.invokeSuper(obj, params);
        }
        return BeanMethodInterceptorManager.doInterceptor(annotationList.toArray(new
Annotation[annotationList.size()]), obj, method, params, methodProxy);

    }

    /**
     *
     * A list of annotated instances needed to assemble the interceptor.
```

```
     *
     * @param annotationList /Full annotation instance list.
     * @param clazz          /The type of the object that the method belongs to.
     * @param method         /Method definition
     */
    private void fillAnnotationList(List<Annotation> annotationList, Class clazz, Method method) {
        Set<Class> classSet = new HashSet<>();
        for (Annotation ann : method.getDeclaredAnnotations()) {
            annotationList.add(ann);
            classSet.add(ann.annotationType());
        }
        for (Annotation ann : clazz.getDeclaredAnnotations()) {
            if (classSet.add(ann.annotationType())) {
                annotationList.add(0, ann);
            }
        }
        for (Annotation ann : clazz.getAnnotations()) {
            if (classSet.add(ann.annotationType())) {
                annotationList.add(0, ann);
            }
        }
    }
}

26:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\core\SpringLiteContext.java
 */
package io.nuls.kernel.lite.core;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.lite.annotation.Autowired;
import io.nuls.kernel.lite.annotation.Component;
import io.nuls.kernel.lite.annotation.Interceptor;
import io.nuls.kernel.lite.annotation.Service;
import io.nuls.kernel.lite.core.bean.InitializingBean;
import io.nuls.kernel.lite.core.interceptor.BeanMethodInterceptor;
import io.nuls.kernel.lite.core.interceptor.BeanMethodInterceptorManager;
import io.nuls.kernel.lite.utils.ScanUtil;
import net.sf.cglib.proxy.Enhancer;
```

```
import net.sf.cglib.proxy.MethodInterceptor;

import java.lang.annotation.Annotation;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

/**
 * ROCspring-framework
 * <p>
 * The simplified version of the ROC framework, referring to the use of the spring-framework,
 * implements a simple dependency injection and aspect-oriented programming for dynamic proxy
implementations.
 *
 * @author Niels Wang
 */
public class SpringLiteContext {

    private static final Map<String, Object> BEAN_OK_MAP = new ConcurrentHashMap<>();
    private static final Map<String, Object> BEAN_TEMP_MAP = new ConcurrentHashMap<>();
    private static final Map<String, Class> BEAN_TYPE_MAP = new ConcurrentHashMap<>();
    private static final Map<Class, Set<String>> CLASS_NAME_SET_MAP = new
ConcurrentHashMap<>();

    private static MethodInterceptor interceptor;

    private static boolean success;

    /**
     * roc
     * Load the roc environment with the default interceptor.
     *
     * @param packName ,The root package of the scan.
     */
    public static void init(final String packName) {
        init(packName, new DefaultMethodInterceptor());
    }

    /**
     * roc
     * Load the roc environment based on the incoming parameters.
```

```
     *
     * @param packName    ,The root package of the scan.
     * @param interceptor ,Method interceptor
     */
    public static void init(final String packName, MethodInterceptor interceptor) {
        SpringLiteContext.interceptor = interceptor;
        List<Class> list = ScanUtil.scan(packName);
        list.forEach((Class clazz) -> checkBeanClass(clazz));
        autowireFields();
        success = true;
    }

    /**
     *
     * Automatically assign values to attributes in an object.
     */
    private static void autowireFields() {
        Set<String> keySet = new HashSet<>(BEAN_TEMP_MAP.keySet());
        for (String key : keySet) {
            try {
                Object bean = BEAN_TEMP_MAP.get(key);
                boolean result = injectionBeanFields(bean, BEAN_TYPE_MAP.get(key));
                if (result) {
                    BEAN_OK_MAP.put(key, bean);
                    BEAN_TEMP_MAP.remove(key);
                    if (bean instanceof InitializingBean) {
                        try {
                            ((InitializingBean) bean).afterPropertiesSet();
                        } catch (Exception e) {
                            Log.error(e);
                        }
                    }
                }
            } catch (Exception e) {
                Log.debug(key + " autowire fields failed!");
            }
        }
    }

    /**
     * Autowired
     * All of the objects tagged with Autowired annotation are injected with dependencies.
```

```
     *
     * @param obj     bean
     * @param objType
     * @throws Exception
     */
    private static boolean injectionBeanFields(Object obj, Class objType) throws Exception {
        Set<Field> fieldSet = getFieldSet(objType);
        boolean result = true;
        for (Field field : fieldSet) {
            boolean b = injectionBeanField(obj, field);
            if (!b) {
                result = true;
            }
        }
        return result;
    }

    //    /**
//     *
//     * Gets all the fields of an object.
//     *
//     * @param objType
//     */
    private static Set<Field> getFieldSet(Class objType) {
        Set<Field> set = new HashSet<>();
        Field[] fields = objType.getDeclaredFields();
        for (Field field : fields) {
            set.add(field);
        }
        if (!objType.getSuperclass().equals(Object.class)) {
            set.addAll(getFieldSet(objType.getSuperclass()));
        }
        return set;
    }

    //    /**
//     * Autowired
//     * Check an attribute of an object, and if the object is marked with Autowired annotations,
//     * it is dependent and will depend on the attribute that is assigned to the object.
//     *
//     * @param obj   bean
//     * @param field
```

```
//     */
    private static boolean injectionBeanField(Object obj, Field field) throws Exception {
        Annotation[] anns = field.getDeclaredAnnotations();
        if (anns == null || anns.length == 0) {
            return true;
        }
        Annotation automired = getFromArray(anns, Autowired.class);
        if (null == automired) {
            return true;
        }
        String name = ((Autowired) automired).value();
        Object value = null;
        if (null == name || name.trim().length() == 0) {
            Set<String> nameSet = CLASS_NAME_SET_MAP.get(field.getType());
            if (nameSet == null || nameSet.isEmpty()) {
                throw new Exception("Can't find the bean,field:" + field.getName());
            } else if (nameSet.size() == 1) {
                name = nameSet.iterator().next();
            } else {
                name = field.getName();
            }
        }
        value = getBean(name);
        if (null == value) {
            throw new Exception("Can't find the bean named:" + name);
        }
        field.setAccessible(true);
        field.set(obj, value);
        field.setAccessible(false);
        return true;
    }

    //    /**
//     * bean
//     * get bean by bean name
//     *
//     * @param name Bean Name
//     */
    private static Object getBean(String name) {
        Object value = BEAN_OK_MAP.get(name);
        if (null == value) {
            value = BEAN_TEMP_MAP.get(name);
```

```
        }
        return value;
    }

    /**
     * Service/Component/Interceptor,bean
     * Check a type, if this is commented on the type annotation, we care about, such as:
(Service/Component/Interceptor), is to load the object, and in the bean manager
     *
     * @param clazz class type
     */
    private static void checkBeanClass(Class clazz) {
        Annotation[] anns = clazz.getDeclaredAnnotations();
        if (anns == null || anns.length == 0) {
            return;
        }
        Annotation ann = getFromArray(anns, Service.class);
        String beanName = null;
        boolean aopProxy = false;
        if (null == ann) {
            ann = getFromArray(anns, Component.class);
            if (null != ann) {
                beanName = ((Component) ann).value();
            }
        } else {
            beanName = ((Service) ann).value();
        }
        if (ann != null) {
            if (beanName == null || beanName.trim().length() == 0) {
                beanName = getBeanName(clazz);
            }
            try {
                loadBean(beanName, clazz, aopProxy);
            } catch (NulsException e) {
                Log.error(e);
                return;
            }
        }
        Annotation interceptorAnn = getFromArray(anns, Interceptor.class);
        if (null != interceptorAnn) {
            BeanMethodInterceptor interceptor = null;
            try {
```

```
                Constructor constructor = clazz.getDeclaredConstructor();
                interceptor = (BeanMethodInterceptor) constructor.newInstance();
            } catch (Exception e) {
                Log.error(e);
                return;
            }
            BeanMethodInterceptorManager.addBeanMethodInterceptor(((Interceptor)
interceptorAnn).value(), interceptor);
        }
    }

    //
//    /**
//     *
//     * Gets the name of the type instance according to the object type.
//     */
    private static String getBeanName(Class clazz) {
        String start = clazz.getSimpleName().substring(0, 1).toLowerCase();
        String end = clazz.getSimpleName().substring(1);
        String beanName = start + end;
        if (BEAN_OK_MAP.containsKey(beanName) || BEAN_TEMP_MAP.containsKey(beanName))
{
            beanName = clazz.getName();
        }
        return beanName;
    }

    /**
     *
     * Gets an instance of the specified annotation type from the array.
     *
     * @param anns  Annotated instance array
     * @param clazz Target annotation type
     * @return Annotation
     */
    private static Annotation getFromArray(Annotation[] anns, Class clazz) {
        for (Annotation ann : anns) {
            if (ann.annotationType().equals(clazz)) {
                return ann;
            }
        }
        return null;
```

```
    }

    //    /**
//     *
//     * Instantiate an instance of this type by instantiating the instantiated object
//     * into the object pool by determining whether the dynamic proxy is used.
//     *
//     * @param beanName
//     * @param clazz
//     * @param proxy
//     */
    private static Object loadBean(String beanName, Class clazz, boolean proxy) throws
NulsException {
        if (BEAN_OK_MAP.containsKey(beanName)) {
            Log.error("bean name repetition (" + beanName + "):" + clazz.getName());
            return BEAN_OK_MAP.get(beanName);
        }
        if (BEAN_TEMP_MAP.containsKey(beanName)) {
            Log.error("bean name repetition (" + beanName + "):" + clazz.getName());
            return BEAN_TEMP_MAP.get(beanName);
        }
        Object bean = null;
        if (proxy) {
            bean = createProxy(clazz, interceptor);
        } else {
            try {
                bean = clazz.newInstance();
            } catch (InstantiationException e) {
                Log.error(e);
                throw new NulsException(e);
            } catch (IllegalAccessException e) {
                Log.error(e);
                throw new NulsException(e);
            }
        }
        BEAN_TEMP_MAP.put(beanName, bean);
        BEAN_TYPE_MAP.put(beanName, clazz);
        addClassNameMap(clazz, beanName);
        return bean;
    }

    //    /**
```

```
//     *
//     * Create an instance of the object using a dynamic proxy.
//     */
    private static Object createProxy(Class clazz, MethodInterceptor interceptor) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(clazz);
        enhancer.setCallback(SpringLiteContext.interceptor);
        return enhancer.create();
    }

    //    /**
//     *
//     * Cache the relationship between the cache type and the instance name.
//     *
//     * @param clazz
//     * @param beanName
//     */
    private static void addClassNameMap(Class clazz, String beanName) {
        Set<String> nameSet = CLASS_NAME_SET_MAP.get(clazz);
        if (null == nameSet) {
            nameSet = new HashSet<>();
        }
        nameSet.add(beanName);
        CLASS_NAME_SET_MAP.put(clazz, nameSet);
        if (null != clazz.getSuperclass() && !clazz.getSuperclass().equals(Object.class)) {
            addClassNameMap(clazz.getSuperclass(), beanName);
        }
        if (clazz.getInterfaces() != null && clazz.getInterfaces().length > 0) {
            for (Class intfClass : clazz.getInterfaces()) {
                addClassNameMap(intfClass, beanName);
            }
        }
    }

    /**
     *
     * Gets the object in the object pool according to the type.
     *
     * @param beanClass
     * @param <T>
     * @return
     */
```

```
    public static <T> T getBean(Class<T> beanClass) {
        Set<String> nameSet = CLASS_NAME_SET_MAP.get(beanClass);
        if (null == nameSet || nameSet.isEmpty()) {
            throw new NulsRuntimeException(KernelErrorCode.DATA_NOT_FOUND);
        }
        if (nameSet.size() > 1) {
            throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
        }
        T value = null;
        String beanName = null;
        for (String name : nameSet) {
            value = (T) BEAN_OK_MAP.get(name);
            beanName = name;
            break;
        }
        if (null == value) {
            value = (T) BEAN_TEMP_MAP.get(beanName);
        }
        return value;
    }

    //    /**
//     *
//     * A managed object is added to the context, which is instantiated using a dynamic proxy based on the incoming type.
//     *
//     * @param clazz
//     */
    public static void putBean(Class clazz) throws NulsException {
        loadBean(getBeanName(clazz), clazz, true);
        autowireFields();
    }

    public static void putBean(Class clazz, boolean proxy) throws NulsException {
        loadBean(getBeanName(clazz), clazz, proxy);
        autowireFields();
    }

    //    /**
//     *
//     * Delete all instances of a type from the context, please call carefully.
//     */
```

```
    public static void removeBean(Class clazz) {
        Set<String> nameSet = CLASS_NAME_SET_MAP.get(clazz);
        if (null == nameSet || nameSet.isEmpty()) {
            return;
        }
        for (String name : nameSet) {
            BEAN_OK_MAP.remove(name);
            BEAN_TEMP_MAP.remove(name);
            BEAN_TYPE_MAP.remove(name);
        }

    }

    //    /**
//     *
//     * Check the status of the instance, and whether the assembly has been completed, that is, all properties are automatically assigned.
//     *
//     * @param bean
//     */
    public static boolean checkBeanOk(Object bean) {
        return BEAN_OK_MAP.containsValue(bean);
    }

    //    /**
//     *
//     * Gets all instances of a type.
//     *
//     * @param beanClass
//     * @param <T>
//     * @return all instances of the type;
//     */
    public static <T> List<T> getBeanList(Class<T> beanClass) throws Exception {
        Set<String> nameSet = CLASS_NAME_SET_MAP.get(beanClass);
        if (null == nameSet || nameSet.isEmpty()) {
            return new ArrayList<>();
        }
        List<T> tlist = new ArrayList<>();
        for (String name : nameSet) {
            T value = (T) BEAN_OK_MAP.get(name);
            if (value == null) {
                value = (T) BEAN_TEMP_MAP.get(name);
```

```
            }
            if (null != value) {
                tlist.add(value);
            }
        }
        return tlist;
    }

    public static boolean isInitSuccess() {
        return success;
    }

    public static Collection<Object> getAllBeanList() {
        return BEAN_OK_MAP.values();
    }
}

27:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\exception\BeanStatusException.java
 */

package io.nuls.kernel.lite.exception;

import io.nuls.kernel.constant.ErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;

/**
 *
 * An exception is thrown when the object instance in the system object pool is not in the right
state.
 *
 * @author Niels
 */
public class BeanStatusException extends NulsRuntimeException {
    public BeanStatusException(ErrorCode message) {
        super(message);
    }

    public BeanStatusException(ErrorCode message, Throwable cause) {
        super(message, cause);
    }
```

```
    public BeanStatusException(Throwable cause) {
        super(cause);
    }

    protected BeanStatusException(ErrorCode message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }

    /*public BeanStatusException(ErrorCode errorCode, String msg) {
        super(errorCode, msg);
    }*/

}

28:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\lite\utils\ScanUtil.java
 */
package io.nuls.kernel.lite.utils;



import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;

import java.io.File;
import java.io.FileFilter;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.URL;
import java.net.URLDecoder;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
 * classpathjarclassclass
 * The utility class is used to scan all classes in the classpath,
 * including the class in the jar file and the class under the folder.
 *
 * @author Niels Wang
```

```
 */
public class ScanUtil {
    private static final ClassLoader CLASS_LOADER = ScanUtil.class.getClassLoader();

    public static final String FILE_TYPE = "file";
    public static final String JAR_TYPE = "jar";
    public static final String CLASS_TYPE = ".class";

    /**
     *
     * Scans all types under the package name.
     *
     * @param packageName “io.nuls”/The package path to be scanned.If the incoming path is
empty, the default is "IO. Nuls"
     * @return /List of all scanned types.
     */
    public static List<Class> scan(String packageName) {
        if (StringUtils.isBlank(packageName)) {
            packageName = "io.nuls";
        }
        List<Class> list = new ArrayList<>();
        Enumeration<URL> dirs;
        try {
            dirs =
Thread.currentThread().getContextClassLoader().getResources(packageName.replace(".", "/"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        if (null == dirs) {
            return list;
        }
        while (dirs.hasMoreElements()) {
            URL url = dirs.nextElement();
            String protocol = url.getProtocol();
            if (FILE_TYPE.equals(protocol)) {
                findClassLocal(packageName, url.getPath(), list);
            } else if (JAR_TYPE.equals(protocol)) {
                findClassJar(packageName, url.getPath(), list);
            }
        }
        return list;
    }
```

```
/**
 *
 * Scan all local types and add the scanned results to the list of types.
 *
 * @param packageName /The package path to be scanned.
 * @param filePath
 * @param list
 */
public static void findClassLocal(final String packageName, String filePath, List<Class> list) {
    File file = null;
    try {
        file = new File(URLDecoder.decode(filePath, "UTF-8"));
    } catch (UnsupportedEncodingException e) {
        Log.error(e);
        return;
    }
    file.listFiles(new LocalFileFilter(packageName, list));
}

/**
 * jar
 *
 * @param packageName /The package path to be scanned.
 * @param pathName    jar/
 * @param list
 */
private static void findClassJar(String packageName, String pathName, List<Class> list) {
    if (StringUtils.isBlank(pathName) || list == null) {
        return;
    }
    JarFile jarFile;
    try {
        int index = pathName.indexOf("!");
        if (index > 0) {
            pathName = pathName.substring(0, index);
        }
        URL url = new URL(pathName);
        jarFile = new JarFile(URLDecoder.decode(url.getPath(), "UTF-8"));
    } catch (IOException e) {
        throw new RuntimeException("could not be parsed as a URI reference");
    }
```

```
    packageName = packageName.replace(".", "/");
    Enumeration<JarEntry> jarEntries = jarFile.entries();
    while (jarEntries.hasMoreElements()) {
        JarEntry jarEntry = jarEntries.nextElement();
        String jarEntryName = jarEntry.getName();
        if (!jarEntryName.contains(packageName) || jarEntryName.equals(packageName + "/")) {
            continue;
        }
        if (jarEntry.isDirectory()) {
            continue;
        } else if (jarEntryName.endsWith(CLASS_TYPE)) {
            Class<?> clazz;
            try {
                String className = jarEntry.getName().replace("/", ".").replace(CLASS_TYPE, "");
                clazz = CLASS_LOADER.loadClass(className);
            } catch (ClassNotFoundException e) {
                continue;
            }
            list.add(clazz);
        }

    }

}

/**
 *
 * File filter
 */
static class LocalFileFilter implements FileFilter {

    private List<Class> list;
    private String packageName;

    public LocalFileFilter(String packageName, List<Class> list) {
        this.list = list;
        this.packageName = packageName;
    }

    @Override
    public boolean accept(File file) {
        if (file.isDirectory()) {
```

```
                findClassLocal(packageName + "." + file.getName(), file.getPath(), list);
                return true;
            }
            if (file.getName().endsWith(CLASS_TYPE)) {
                Class<?> clazz;
                try {
                    clazz = CLASS_LOADER.loadClass(packageName + "." +
file.getName().replace(CLASS_TYPE, ""));
                } catch (ClassNotFoundException e) {
                    return false;
                }
                list.add(clazz);
                return true;
            }
            return false;
        }
    }
}

29:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\MicroKernelBootstrap.java
 */
package io.nuls.kernel;

import io.nuls.core.tools.cfg.ConfigLoader;
import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.func.TimeService;
import io.nuls.kernel.i18n.I18nUtils;
import io.nuls.kernel.lite.core.ModularServiceMethodInterceptor;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.utils.TransactionManager;
import io.nuls.kernel.validate.ValidatorManager;

import java.io.IOException;
import java.lang.reflect.Field;

/**
 * @author Niels
```

```
 */
public class MicroKernelBootstrap extends BaseModuleBootstrap {
    private static final MicroKernelBootstrap INSTANCE = new MicroKernelBootstrap();

    private MicroKernelBootstrap() {
        super(NulsConstant.MODULE_ID_MICROKERNEL);
    }

    public static MicroKernelBootstrap getInstance() {
        return INSTANCE;
    }

    /**
     * spring-lite
     * <p>
     * Micro kernel module initialization method, for the operation are: load profile information into memory,
     * set the default encoding and language, start the spring - lite container, and start the version manager and validator manager
     */
    @Override
    public void init() {
        try {
            NulsConfig.NULS_CONFIG = ConfigLoader.loadIni(NulsConstant.USER_CONFIG_FILE);
            NulsConfig.MODULES_CONFIG = ConfigLoader.loadIni(NulsConstant.MODULES_CONFIG_FILE);
        } catch (IOException e) {
            Log.error("Client start failed", e);
            throw new RuntimeException("Client start failed");
        }

        TimeService.getInstance().start();

        //set system language
        try {
            NulsConfig.DEFAULT_ENCODING = NulsConfig.NULS_CONFIG.getCfgValue(NulsConstant.CFG_SYSTEM_SECTION, NulsConstant.CFG_SYSTEM_DEFAULT_ENCODING);
            String language = NulsConfig.NULS_CONFIG.getCfgValue(NulsConstant.CFG_SYSTEM_SECTION, NulsConstant.CFG_SYSTEM_LANGUAGE);
            I18nUtils.setLanguage(language);
```

```
    } catch (Exception e) {
        Log.error(e);
    }
    SpringLiteContext.init("io.nuls", new ModularServiceMethodInterceptor());
    try {
        NulsConfig.VERSION = getKernelVersion();
        TransactionManager.init();
        ValidatorManager.init();
    } catch (Exception e) {
        Log.error(e);
    }
}

@Override
public void start() {
}

@Override
public void shutdown() {
}

@Override
public void destroy() {
}

@Override
public String getInfo() {
    StringBuilder info = new StringBuilder();
    info.append("kernel module:\n");

    String version = getKernelVersion();
    if (StringUtils.isBlank(version)) {
        info.append("module-version:");
        info.append(version);
    }
    info.append("\nDEFAULT_ENCODING:");
    info.append(NulsConfig.DEFAULT_ENCODING);
    info.append("\nLanguage");
    info.append(I18nUtils.getLanguage());
    info.append("\nNuls-Config:");
    info.append(NulsConfig.NULS_CONFIG);
    info.append("\nModule-Config:");
```

```
        info.append(NulsConfig.MODULES_CONFIG);
        return info.toString();
    }

    private String getKernelVersion() {
        try {
            Class mavenInfo = Class.forName("io.nuls.module.version.KernelMavenInfo");
            Field field = mavenInfo.getDeclaredField("VERSION");
            return (String) field.get(mavenInfo);
        } catch (Exception e) {
            //do nothing
        }
        return "0.0.0";
    }
}

30:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\Address.java
 */

package io.nuls.kernel.model;

import io.nuls.core.tools.array.ArraysTool;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.SerializeUtils;

/**
 * @author: Chralie
 */
public class Address {

    /**
     * hash length
     */
    public static final int ADDRESS_LENGTH = 23;

    /**
     * RIPEMD160 length
```

```
     */
    private static final int LENGTH = 20;

    /**
     * chain id
     */
    private short chainId = NulsContext.DEFAULT_CHAIN_ID;

    /**
     * address type
     */
    private byte addressType;

    /**
     * hash160 of public key
     */
    protected byte[] hash160;

    protected byte[] addressBytes;

    //    /**
//     * @param address
//     */
    public Address(String address) {
        try {
            byte[] bytes = AddressTool.getAddress(address);

            Address addressTmp = Address.fromHashs(bytes);
            this.chainId = addressTmp.getChainId();
            this.addressType = addressTmp.getAddressType();
            this.hash160 = addressTmp.getHash160();
            this.addressBytes = calcAddressbytes();
        } catch (Exception e) {
            Log.error(e);
        }
    }

    public Address(short chainId, byte addressType, byte[] hash160) {
        this.chainId = chainId;
        this.addressType = addressType;
        this.hash160 = hash160;
        this.addressBytes = calcAddressbytes();
```

```
}

public byte[] getHash160() {
    return hash160;
}


public short getChainId() {
    return chainId;
}

public static Address fromHashs(String address) throws Exception {
    byte[] bytes = AddressTool.getAddress(address);
    return fromHashs(bytes);
}

public static Address fromHashs(byte[] hashs) {
    if (hashs == null || hashs.length != ADDRESS_LENGTH) {
        throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
    }

    short chainId = SerializeUtils.bytes2Short(hashs);
    byte addressType = hashs[2];
    byte[] content = new byte[LENGTH];
    System.arraycopy(hashs, 3, content, 0, LENGTH);

    Address address = new Address(chainId, addressType, content);
    return address;
}

public byte[] calcAddressbytes() {
    byte[] body = new byte[ADDRESS_LENGTH];
    System.arraycopy(SerializeUtils.shortToBytes(chainId), 0, body, 0, 2);
    body[2] = this.addressType;
    System.arraycopy(hash160, 0, body, 3, hash160.length);
    return body;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
```

```
        }
        if (obj instanceof Address) {
            Address other = (Address) obj;
            return ArraysTool.arrayEquals(this.addressBytes, other.getAddressBytes());
        }
        return false;
    }

    public byte[] getAddressBytes() {
        return addressBytes;
    }

    public void setAddressBytes(byte[] addressBytes) {
        this.addressBytes = addressBytes;
    }

    public byte getAddressType() {
        return addressType;
    }

    public void setAddressType(byte addressType) {
        this.addressType = addressType;
    }

    public static int size() {
        return ADDRESS_LENGTH;
    }

    @Override
    public String toString() {
        return AddressTool.getStringAddressByBytes(this.addressBytes);
    }

    public String getBase58() {
        return AddressTool.getStringAddressByBytes(this.addressBytes);
    }
}

31:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\BaseNulsData.java
 */
package io.nuls.kernel.model;
```

```
import io.nuls.core.tools.crypto.UnsafeByteArrayOutputStream;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.exception.NulsVerificationException;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.validate.ValidateResult;
import io.nuls.kernel.validate.ValidatorManager;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.Serializable;
import java.util.Arrays;

/**
 * @author Niels
 */
public abstract class BaseNulsData implements NulsData, Serializable, Cloneable {

    @Override
    public final byte[] serialize() throws IOException {
        ByteArrayOutputStream bos = null;
        try {
            int size = size();
            bos = new UnsafeByteArrayOutputStream(size);
            NulsOutputStreamBuffer buffer = new NulsOutputStreamBuffer(bos);
            if (size == 0) {
                bos.write(NulsConstant.PLACE_HOLDER);
            } else {
                serializeToStream(buffer);
            }
            byte[] bytes = bos.toByteArray();
            if (bytes.length != this.size()) {
                throw new NulsRuntimeException(KernelErrorCode.SERIALIZE_ERROR);
            }
            return bytes;
        } finally {
            if (bos != null) {
                try {
```

```
                bos.close();
            } catch (IOException e) {
                throw e;
            }
        }
    }
}

protected abstract void serializeToStream(NulsOutputStreamBuffer stream) throws
IOException;


@Override
public final void parse(byte[] bytes, int cursor) throws NulsException {
    if (bytes == null || bytes.length == 0 || ((bytes.length == 4) &&
Arrays.equals(NulsConstant.PLACE_HOLDER, bytes))) {
        return;
    }
    NulsByteBuffer byteBuffer = new NulsByteBuffer(bytes);
    byteBuffer.setCursor(cursor);
    this.parse(byteBuffer);
}

public abstract void parse(NulsByteBuffer byteBuffer) throws NulsException;

public final ValidateResult verify() {
    ValidateResult result = ValidatorManager.startDoValidator(this);
    return result;
}

public final void verifyWithException() throws NulsVerificationException {
    ValidateResult result = this.verify();
    if (result.isFailed()) {
        throw new NulsVerificationException(result.getErrorCode());
    }
}

}

32:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\Block.java
 */
```

```
package io.nuls.kernel.model;

import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.TransactionManager;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * @author win10
 */
public class Block extends BaseNulsData implements Cloneable {

    private BlockHeader header;
    private List<Transaction> txs;

    @Override
    public int size() {
        int size = header.size();
        for (Transaction tx : txs) {
            size += tx.size();
        }
        return size;
    }

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        header.serializeToStream(stream);
        for (Transaction tx : txs) {
            stream.write(tx.serialize());
        }
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        header = new BlockHeader();
        header.parse(byteBuffer);
```

```
        try {
            txs = TransactionManager.getInstances(byteBuffer, header.getTxCount());
        } catch (Exception e) {
            throw new NulsRuntimeException(KernelErrorCode.DESERIALIZE_ERROR);
        }
        for (Transaction tx : txs) {
            tx.setBlockHeight(header.getHeight());
        }
    }

    public List<Transaction> getTxs() {
        return txs;
    }

    public void setTxs(List<Transaction> txs) {
        this.txs = txs;
    }

    public BlockHeader getHeader() {
        return header;
    }

    public void setHeader(BlockHeader header) {
        this.header = header;
    }

    //    /**
//     * hash
//     * Loop through the list of trades to remove all of the trading hash, in the same order as the list of transactions.
//     */
    public List<NulsDigestData> getTxHashList() {
        List<NulsDigestData> list = new ArrayList<>();
        for (Transaction tx : txs) {
            if (null == tx) {
                continue;
            }
            list.add(tx.getHash());
        }
        return list;
    }
```

```
}

33:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\BlockHeader.java
 */
package io.nuls.kernel.model;

import io.nuls.core.tools.crypto.Hex;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.script.BlockSignature;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;

/**
 * @author vivi
 */
public class BlockHeader extends BaseNulsData {

    private transient NulsDigestData hash;
    private NulsDigestData preHash;
    private NulsDigestData merkleHash;
    private long time;
    private long height;
    private long txCount;
    private BlockSignature blockSignature;
    private byte[] extend;
    /**
     * pierre add
     */
    private transient byte[] stateRoot;

    private transient int size;
    private transient byte[] packingAddress;

    public BlockHeader() {
```

```
}

protected synchronized void calcHash() {
    if (null != this.hash) {
        return;
    }
    hash = forceCalcHash();
}

@Override
public int size() {
    int size = 0;
    size += SerializeUtils.sizeOfNulsData(preHash);
    size += SerializeUtils.sizeOfNulsData(merkleHash);
    size += SerializeUtils.sizeOfUint48();
    size += SerializeUtils.sizeOfUint32();
    size += SerializeUtils.sizeOfUint32();
    size += SerializeUtils.sizeOfBytes(extend);
    size += SerializeUtils.sizeOfNulsData(blockSignature);
    return size;
}

@Override
protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
    stream.writeNulsData(preHash);
    stream.writeNulsData(merkleHash);
    stream.writeUint48(time);
    stream.writeUint32(height);
    stream.writeUint32(txCount);
    stream.writeBytesWithLength(extend);
    stream.writeNulsData(blockSignature);
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    this.preHash = byteBuffer.readHash();
    this.merkleHash = byteBuffer.readHash();
    this.time = byteBuffer.readUint48();
    this.height = byteBuffer.readUint32();
    this.txCount = byteBuffer.readUint32();
    this.extend = byteBuffer.readByLengthByte();
    try {
```

```
            this.hash = NulsDigestData.calcDigestData(this.serialize());
        } catch (IOException e) {
            Log.error(e);
        }
        this.blockSignature = byteBuffer.readNulsData(new BlockSignature());
//        }
    }

    private NulsDigestData forceCalcHash() {
        try {
            BlockHeader header = (BlockHeader) this.clone();
            header.setBlockSignature(null);
            return NulsDigestData.calcDigestData(header.serialize());
        } catch (Exception e) {
            throw new NulsRuntimeException(e);
        }
    }

    public NulsDigestData getHash() {
        if (null == hash) {
            calcHash();
        }
        return hash;
    }

    public void setHash(NulsDigestData hash) {
        this.hash = hash;
    }

    public NulsDigestData getPreHash() {
        return preHash;
    }

    public void setPreHash(NulsDigestData preHash) {
        this.preHash = preHash;
    }

    public NulsDigestData getMerkleHash() {
        return merkleHash;
    }

    public void setMerkleHash(NulsDigestData merkleHash) {
```

```
    this.merkleHash = merkleHash;
}

public long getTime() {
    return time;
}

public void setTime(long time) {
    this.time = time;
}

public long getHeight() {
    return height;
}

public void setHeight(long height) {
    this.height = height;
}

public long getTxCount() {
    return txCount;
}

public void setTxCount(long txCount) {
    this.txCount = txCount;
}

public BlockSignature getBlockSignature() {
    return blockSignature;
}

public void setBlockSignature(BlockSignature scriptSign) {
    this.blockSignature = scriptSign;
}

public byte[] getPackingAddress() {
    if (null == packingAddress && this.blockSignature != null) {
        this.packingAddress = AddressTool.getAddress(blockSignature.getPublicKey());
    }
    return packingAddress;
}
```

```
public byte[] getExtend() {
    return extend;
}

public void setExtend(byte[] extend) {
    this.extend = extend;
}

public int getSize() {
    return size;
}

public void setSize(int size) {
    this.size = size;
}

public void setPackingAddress(byte[] packingAddress) {
    this.packingAddress = packingAddress;
}

public byte[] getStateRoot() {
    return stateRoot;
}

public void setStateRoot(byte[] stateRoot) {
    this.stateRoot = stateRoot;
}

@Override
public String toString() {
    return "BlockHeader{" +
            "hash=" + hash.getDigestHex() +
            ", preHash=" + preHash.getDigestHex() +
            ", merkleHash=" + merkleHash.getDigestHex() +
            ", time=" + time +
            ", height=" + height +
            ", txCount=" + txCount +
            ", blockSignature=" + blockSignature +
            //", extend=" + Arrays.toString(extend) +
            ", size=" + size +
            ", packingAddress=" + (packingAddress == null ? packingAddress :
```

```
AddressTool.getStringAddressByBytes(packingAddress)) +
                '}';
    }
}
```

34:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\model\Coin.java

```
 *
 */

package io.nuls.kernel.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.func.TimeService;
import io.nuls.kernel.script.Script;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;

/**
 * @author ln
 */
public class Coin extends BaseNulsData {

    private byte[] owner;

    private Na na;

    private long lockTime;

    private transient Coin from;
    /**
     * CoinData
     */
    private transient String key;
```

```
private transient byte[] tempOwner;

public Coin() {
}

public Coin(byte[] owner, Na na) {
    this.owner = owner;
    this.na = na;
}

public Coin(byte[] owner, Na na, long lockTime) {
    this(owner, na);
    this.lockTime = lockTime;
}

@Override
protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
    stream.writeBytesWithLength(owner);
    stream.writeInt64(na.getValue());
    stream.writeUint48(lockTime);
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    this.owner = byteBuffer.readByLengthByte();
    this.na = Na.valueOf(byteBuffer.readInt64());
    this.lockTime = byteBuffer.readUint48();
}

@Override
public int size() {
    int size = 0;
    size += SerializeUtils.sizeOfBytes(owner);
    size += SerializeUtils.sizeOfInt64();
    size += SerializeUtils.sizeOfUint48();
    return size;
}


public Na getNa() {
    return na;
```

```
}

public byte[] getOwner() {
    return owner;
}

public long getLockTime() {
    return lockTime;
}

public Coin getFrom() {
    return from;
}

public void setFrom(Coin from) {
    this.from = from;
}

public void setOwner(byte[] owner) {
    this.owner = owner;
}

public void setNa(Na na) {
    this.na = na;
}

public void setLockTime(long lockTime) {
    this.lockTime = lockTime;
}

public String getKey() {
    return key;
}

public Coin setKey(String key) {
    this.key = key;
    return this;
}

public byte[] getTempOwner() {
    return tempOwner;
}
```

```
public void setTempOwner(byte[] tempOwner) {
    this.tempOwner = tempOwner;
}

/**
 * coin
 *
 * @return boolean
 */
public boolean usable() {
    long bestHeight = NulsContext.getInstance().getBestHeight();
    return usable(bestHeight);
}

public boolean usable(Long bestHeight) {
    if (lockTime < 0) {
        return false;
    }
    if (lockTime == 0) {
        return true;
    }

    long currentTime = TimeService.currentTimeMillis();
    //long bestHeight = NulsContext.getInstance().getBestHeight();

    if (lockTime > NulsConstant.BlOCKHEIGHT_TIME_DIVIDE) {
        if (lockTime <= currentTime) {
            return true;
        } else {
            return false;
        }
    } else {
        if (lockTime <= bestHeight) {
            return true;
        } else {
            return false;
        }
    }
}

@Override
```

```
    public String toString() {
        return "Coin{" +
                "owner=" + AddressTool.getStringAddressByBytes(owner) +
                ", na=" + na.getValue() +
                ", lockTime=" + lockTime +
                ", from=" + from +
                ", key='" + key + '\'' +
                '}';
    }

    @JsonIgnore
    public byte[] getAddress() {
        byte[] address = new byte[23];
        //ownerowner
        if (owner == null || owner.length == 23) {
            return owner;
        } else {
            Script scriptPubkey = new Script(owner);
            //P2PKH23
            if (scriptPubkey.isSentToAddress()) {
                System.arraycopy(owner, 3, address, 0, 23);
            }
            //P2SHmultiUTXO23
            else if (scriptPubkey.isPayToScriptHash()) {
                scriptPubkey.isSentToMultiSig();
                System.arraycopy(owner, 2, address, 0, 23);
            }else{
                throw new
NulsRuntimeException(KernelErrorCode.ADDRESS_IS_NOT_BELONGS_TO_CHAIN);
            }
        }
        return address;
    }
}

35:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\CoinData.java
 *
 */

package io.nuls.kernel.model;
```

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.script.Script;
import io.nuls.kernel.script.SignatureUtil;
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.*;

/**
 * @author ln
 */
public class CoinData extends BaseNulsData {

    private List<Coin> from;

    private List<Coin> to;

    public CoinData() {
        from = new ArrayList<>();
        to = new ArrayList<>();
    }

    /**
     * serialize important field
     */
    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        int fromCount = from == null ? 0 : from.size();
        stream.writeVarInt(fromCount);
        if (null != from) {
            for (Coin coin : from) {
                stream.writeNulsData(coin);
            }
        }
        int toCount = to == null ? 0 : to.size();
        stream.writeVarInt(toCount);
        if (null != to) {
            for (Coin coin : to) {
```

```
            stream.writeNulsData(coin);
        }
    }
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    int fromCount = (int) byteBuffer.readVarInt();

    if (0 < fromCount) {
        List<Coin> from = new ArrayList<>();
        for (int i = 0; i < fromCount; i++) {
            from.add(byteBuffer.readNulsData(new Coin()));
        }
        this.from = from;
    }

    int toCount = (int) byteBuffer.readVarInt();

    if (0 < toCount) {
        List<Coin> to = new ArrayList<>();
        for (int i = 0; i < toCount; i++) {
            to.add(byteBuffer.readNulsData(new Coin()));
        }
        this.to = to;
    }
}

@Override
public int size() {
    int size = SerializeUtils.sizeOfVarInt(from == null ? 0 : from.size());
    if (null != from) {
        for (Coin coin : from) {
            size += SerializeUtils.sizeOfNulsData(coin);
        }
    }
    size += SerializeUtils.sizeOfVarInt(to == null ? 0 : to.size());
    if (null != to) {
        for (Coin coin : to) {
            size += SerializeUtils.sizeOfNulsData(coin);
        }
    }
```

```
    return size;
}

public List<Coin> getFrom() {
    return from;
}

public void setFrom(List<Coin> from) {
    this.from = from;
}

public List<Coin> getTo() {
    return to;
}

public void setTo(List<Coin> to) {
    this.to = to;
}

/**
 *
 * The handling charge for the transaction.
 *
 * @return tx fee
 */
@JsonIgnore
public Na getFee() {
    Na toNa = Na.ZERO;
    for (Coin coin : to) {
        toNa = toNa.add(coin.getNa());
    }
    Na fromNa = Na.ZERO;
    for (Coin coin : from) {
        fromNa = fromNa.add(coin.getNa());
    }
    return fromNa.subtract(toNa);
}

public void addTo(Coin coin) {
    if (null == to) {
        to = new ArrayList<>();
    }
```

```
        if(coin.getOwner().length == 23 && coin.getOwner()[2] ==
NulsContext.P2SH_ADDRESS_TYPE){
            Script scriptPubkey = SignatureUtil.createOutputScript(coin.getOwner());
            coin.setOwner(scriptPubkey.getProgram());
        }
        to.add(coin);
    }

    public void addFrom(Coin coin) {
        if (null == from) {
            from = new ArrayList<>();
        }
        from.add(coin);
    }

    @JsonIgnore
    public Set<byte[]> getAddresses() {
        Set<byte[]> addressSet = new HashSet<>();
        if (to != null && to.size() != 0) {
            for (int i = 0; i < to.size(); i++) {
                byte[] owner = to.get(i).getAddress();
                boolean hasExist = false;
                for (byte[] address : addressSet) {
                    //todo 20180919
                    if (Arrays.equals(owner, address)) {
                        hasExist = true;
                        break;
                    }
                }
                if (!hasExist) {
                    addressSet.add(owner);
                }
            }
        }
        return addressSet;
    }
}

36:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\CommandResult.java
 *
 */
```

```
package io.nuls.kernel.model;

import io.nuls.core.tools.json.JSONUtils;
import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;

import java.util.Map;

/**
 * @author Niels
 */
public class CommandResult {

    public boolean success;

    private String message;

    public boolean isSuccess() {
        return success;
    }

    public CommandResult setSuccess(boolean success) {
        this.success = success;
        return this;
    }

    public String getMessage() {
        return message;
    }

    public CommandResult setMessage(String message) {
        this.message = message;
        return this;
    }

    @Override
    public String toString() {
        if (StringUtils.isBlank(message)) {
            return "result:" + success;
        } else {
            return message;
```

```
    }
}

public static CommandResult getFailed(String message) {
    CommandResult result = new CommandResult();
    result.setMessage(message);
    result.setSuccess(false);
    return result;
}

public static CommandResult getFailed(RpcClientResult rpcResult) {
    CommandResult result = new CommandResult();
    Map<String, Object> map = (Map) rpcResult.getData();
    result.setMessage((String) map.get("msg"));
    result.setSuccess(false);
    return result;
}

public static CommandResult getResult(RpcClientResult rpcResult) {
    if (null == rpcResult) {
        return CommandResult.getFailed("Result is null!");
    }
    CommandResult result = new CommandResult();
    result.setSuccess(rpcResult.isSuccess());
    String message = "";
    if (!rpcResult.isSuccess()) {
        Map<String, Object> map = (Map) rpcResult.getData();
        message = (String) map.get("msg");
        //message += ":";
    } else {
        try {
            message += JSONUtils.obj2PrettyJson(rpcResult.getData());
        } catch (Exception e) {
            Log.error(e);
        }
    }
    result.setMessage(message);
    return result;
}

public static CommandResult getSuccess(String message) {
    return new CommandResult().setSuccess(true).setMessage(message);
```

```
    }


    public static RpcClientResult dataTransformValue(RpcClientResult rpcResult) {
        Map<String, Object> map = ((Map) rpcResult.getData());
        if (null != map) {
            rpcResult.setData(map.get("value"));
        }
        return rpcResult;
    }


    public static RpcClientResult dataMultiTransformValue(RpcClientResult rpcResult) {
        Map<String, Object> map = ((Map) rpcResult.getData());
        if (null != map) {
            rpcResult.setData(map.get("txData"));
        }
        return rpcResult;
    }

    public static RpcClientResult dataTransformList(RpcClientResult rpcResult) {
        Map<String, Object> map = ((Map) rpcResult.getData());
        if (null != map) {
            rpcResult.setData(map.get("list"));
        }
        return rpcResult;
    }

}

37:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\ErrorData.java
 */
package io.nuls.kernel.model;

import io.nuls.kernel.constant.ErrorCode;

/**
 * @author: Charlie
 */
public class ErrorData {
```

```
    private String code;

    private String msg;

    public ErrorData() {
    }

    public ErrorData(String code, String msg) {
        this.code = code;
        this.msg = msg;
    }

    public ErrorData(ErrorCode errorCode) {
        this.code = errorCode.getCode();
        this.msg = errorCode.getMsg();
    }

    public static ErrorData getErrorData(ErrorCode errorCode) {
        return new ErrorData(errorCode);
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}

38:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\model\Na.java
 */
package io.nuls.kernel.model;
```

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.core.tools.calc.LongUtils;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;

import java.io.Serializable;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * Nuls unit
 *
 * @author Niels
 */
public final class Na implements Comparable<Na>, Serializable {

    private static final long serialVersionUID = 69781492023344275 37L;

    public static final int SMALLEST_UNIT_EXPONENT = 8;

    private static final NumberFormat numberFormat = new DecimalFormat("###.00######");

    public static final long NA_VALUE = (long) Math.pow(10, SMALLEST_UNIT_EXPONENT);

    public static final long TOTAL_VALUE = 100000000L;
    public static final long MAX_NA_VALUE = LongUtils.mul(TOTAL_VALUE, ((long) Math.pow(10,
SMALLEST_UNIT_EXPONENT)));

    /**
     * Total amount of token
     */
    public static final Na MAX = Na.valueOf(TOTAL_VALUE).multiply(NA_VALUE);

    /**
     * 0 Nuls
     */
    public static final Na ZERO = Na.valueOf(0);

    /**
```

```
 * 1 Nuls
 */
public static final Na NA = Na.valueOf(NA_VALUE);

/**
 * 0.01 Nuls
 */
public static final Na CENT = NA.divide(100);

/**
 * 0.001 Nuls
 */
public static final Na MILLICOIN = NA.divide(1000);

/**
 * 0.000001 Nuls
 */
public static final Na MICROCOIN = MILLICOIN.divide(1000);

/**
 * amount
 */
private final long value;

private Na(final long na) {
    if (MAX_NA_VALUE < na || na < 0) {
        throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
    }
    this.value = na;
}

public static Na valueOf(final long na) {
    if (MAX_NA_VALUE < na) {
        throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
    }
    return new Na(na);
}

public int smallestUnitExponent() {
    return SMALLEST_UNIT_EXPONENT;
}
```

```
    public long getValue() {
        return value;
    }

    public static Na parseNuls(final String str) {
        try {
            long value = new
BigDecimal(str).movePointRight(SMALLEST_UNIT_EXPONENT).setScale(SMALLEST_UNIT_EX
PONENT, RoundingMode.HALF_DOWN).longValue();
            return Na.valueOf(value);
        } catch (ArithmeticException e) {
            throw new IllegalArgumentException(e);
        }
    }

    public static Na parseNuls(final double nuls) {
        try {
            long value = new
BigDecimal(nuls).movePointRight(SMALLEST_UNIT_EXPONENT).setScale(SMALLEST_UNIT_E
XPONENT, RoundingMode.HALF_DOWN).longValue();
            return Na.valueOf(value);
        } catch (ArithmeticException e) {
            throw new IllegalArgumentException(e);
        }
    }

    public double toDouble() {
        return new
BigDecimal(this.value).movePointLeft(SMALLEST_UNIT_EXPONENT).setScale(SMALLEST_UNI
T_EXPONENT, RoundingMode.HALF_DOWN).doubleValue();
    }

    public Na add(final Na value) {
        return new Na(LongUtils.add(this.value, value.value));
    }

    public Na plus(final Na value) {
        return add(value);
    }

    public Na subtract(final Na value) {
        return new Na(LongUtils.sub(this.value, value.value));
```

```
    }

    public Na minus(final Na value) {
        return subtract(value);
    }

    public Na multiply(final long factor) {
        return new Na(LongUtils.mul(this.value, factor));
    }

    public Na times(final long factor) {
        return multiply(factor);
    }

    public Na times(final int factor) {
        return multiply(factor);
    }

    public Na divide(final long divisor) {
        return new Na(LongUtils.div(this.value, divisor));
    }

    public Na div(final long divisor) {
        return divide(divisor);
    }

    public Na div(final int divisor) {
        return divide(divisor);
    }

    public Na[] divideAndRemainder(final long divisor) {
        return new Na[]{new Na(LongUtils.div(this.value, divisor)), new Na(LongUtils.mod(this.value,
divisor))};
    }

    public long divide(final Na divisor) {
        return LongUtils.div(this.value, divisor.value);
    }

    //
//    /**
//     * Returns true if and only if this instance represents a monetary value greater than zero,
```

```
//     * otherwise false.
//     */
    @JsonIgnore
    public boolean isPositive() {
        return signum() == 1;
    }

    //
//     /**
//     * Returns true if and only if this instance represents a monetary value less than zero,
//     * otherwise false.
//     */
    @JsonIgnore
    public boolean isNegative() {
        return signum() == -1;
    }

    //     /**
//     * Returns true if and only if this instance represents zero monetary value,
//     * otherwise false.
//     */
    @JsonIgnore
    public boolean isZero() {
        return signum() == 0;
    }

    //     /**
//     * Returns true if the monetary value represented by this instance is greater than that
//     * of the given other Na, otherwise false.
//     */
    public boolean isGreaterThan(Na other) {
        return compareTo(other) > 0;
    }

    public boolean isGreaterOrEquals(Na other) {
        return compareTo(other) >= 0;
    }

    //     /**
//     * Returns true if the monetary value represented by this instance is less than that
//     * of the given other Na, otherwise false.
//     */
```

```
public boolean isLessThan(Na other) {
    return compareTo(other) < 0;
}

public Na shiftLeft(final int n) {
    return new Na(this.value << n);
}

public Na shiftRight(final int n) {
    return new Na(this.value >> n);
}

public int signum() {
    if (this.value == 0) {
        return 0;
    }
    return this.value < 0 ? -1 : 1;
}

public Na negate() {
    return new Na(-this.value);
}

public String toText() {

    BigDecimal amount = new BigDecimal(value).divide(BigDecimal.valueOf(Na.NA.value));
    return amount.toPlainString();
}

@Override
public String toString() {
    return Long.toString(value);
}

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

```
        return this.value == ((Na) o).value;
    }

    @Override
    public int hashCode() {
        return (int) this.value;
    }

    @Override
    public int compareTo(final Na other) {
        if (other == null) {
            return -1;
        }
        return Long.compare(this.value, other.value);
    }

    public String toCoinString() {
//        double d = new BigDecimal(value).movePointLeft(8).doubleValue();
//        return numberFormat.format(d);
        return toText();
    }

    //    /**
//     * Long  Integer Na  NUSL(double)
//     * NUSL
//     * @param object
//     * @return
//     */
    public static double naToNuls(Object object) {
        if (null == object) {
            return 0;
        }
        Long na = null;
        if (object instanceof Long) {
            na = (Long) object;
        } else if (object instanceof Integer) {
            na = ((Integer) object).longValue();
        } else if (object instanceof Double) {
            return (Double) object;
        } else if (object instanceof Float) {
            return Double.parseDouble(String.valueOf(object));
        } else {
```

```
            return 0;
        }
        return (Na.valueOf(na)).toDouble();
    }
}
```

39:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\model\NulsData.java

```
 */

package io.nuls.kernel.model;

import io.nuls.kernel.exception.NulsException;

import java.io.IOException;

/**
 * @author: Niels Wang
 */
public interface NulsData {

    int size();

    byte[] serialize() throws IOException;

    void parse(byte[] bytes, int cursor) throws NulsException;
}
```

40:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\model\NulsDigestData.java

```
 */
package io.nuls.kernel.model;


import io.nuls.core.tools.crypto.Hex;
import io.nuls.core.tools.crypto.Sha256Hash;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.exception.NulsException;
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.Arrays;
import java.util.HashMap;
import java.util.List;

/**
 * @author facjas
 */
public class NulsDigestData extends BaseNulsData {

    public static final int HASH_LENGTH = 34;

    protected byte digestAlgType = DIGEST_ALG_SHA256;

    protected byte[] digestBytes;

    public static byte DIGEST_ALG_SHA256 = 0;
    public static byte DIGEST_ALG_SHA160 = 1;

    public NulsDigestData() {
    }

    public NulsDigestData(byte algType, byte[] bytes) {
        this.digestBytes = bytes;
        this.digestAlgType = algType;
    }

    @Override
    public int size() {
        return SerializeUtils.sizeOfBytes(digestBytes) + 1;
    }

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer buffer) throws IOException {
        buffer.write(digestAlgType);
        buffer.writeBytesWithLength(digestBytes);
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        digestAlgType = byteBuffer.readByte();
        this.digestBytes = byteBuffer.readByLengthByte();
```

```
}

public byte getDigestAlgType() {
    return digestAlgType;
}

public void setDigestAlgType(byte digestAlgType) {
    this.digestAlgType = digestAlgType;
}


public String getDigestHex() {
    try {
        return Hex.encode(serialize());
    } catch (IOException e) {
        Log.error(e);
        return null;
    }
}

public static NulsDigestData fromDigestHex(String hex) throws NulsException {
    byte[] bytes = Hex.decode(hex);
    NulsDigestData hash = new NulsDigestData();
    hash.parse(bytes, 0);
    return hash;
}

public static boolean validHash(String hex) {
    try {
        fromDigestHex(hex);
        return true;
    } catch (Exception e) {
        return false;
    }
}

public static NulsDigestData calcDigestData(BaseNulsData data) {
    return calcDigestData(data, (byte) 0);
}

public static NulsDigestData calcDigestData(BaseNulsData data, byte digestAlgType) {
    try {
```

```
            return calcDigestData(data.serialize(), digestAlgType);
        } catch (Exception e) {
            Log.error(e);
            return null;
        }
    }

    public byte[] getDigestBytes() {
        return digestBytes;
    }

    public static NulsDigestData calcDigestData(byte[] data) {
        return calcDigestData(data, (byte) 0);
    }

    public static NulsDigestData calcDigestData(byte[] data, byte digestAlgType) {
        NulsDigestData digestData = new NulsDigestData();
        digestData.setDigestAlgType(digestAlgType);
        if ((byte) 0 == digestAlgType) {
            byte[] content = Sha256Hash.hashTwice(data);
            digestData.digestBytes = content;
            return digestData;
        }
        //todo extend other algType
        if ((byte) 1 == digestAlgType) {
            byte[] content = SerializeUtils.sha256hash160(data);
            digestData.digestBytes = content;
            return digestData;
        }
        return null;
    }

    public static NulsDigestData calcMerkleDigestData(List<NulsDigestData> ddList) {
        int levelOffset = 0;
        for (int levelSize = ddList.size(); levelSize > 1; levelSize = (levelSize + 1) / 2) {
            for (int left = 0; left < levelSize; left += 2) {
                int right = Math.min(left + 1, levelSize - 1);
                byte[] leftBytes = SerializeUtils.reverseBytes(ddList.get(levelOffset +
left).getDigestBytes());
                byte[] rightBytes = SerializeUtils.reverseBytes(ddList.get(levelOffset +
right).getDigestBytes());
                byte[] whole = new byte[leftBytes.length + rightBytes.length];
```

```
            System.arraycopy(leftBytes, 0, whole, 0, leftBytes.length);
            System.arraycopy(rightBytes, 0, whole, leftBytes.length, rightBytes.length);
            NulsDigestData digest = NulsDigestData.calcDigestData(whole);
            ddList.add(digest);
        }
        levelOffset += levelSize;
    }
    byte[] bytes = ddList.get(ddList.size() - 1).getDigestBytes();
    Sha256Hash merkleHash = Sha256Hash.wrap(bytes);
    NulsDigestData digestData = new NulsDigestData();
    digestData.digestBytes = merkleHash.getBytes();
    return digestData;
}

/**
 * Indicates whether some other object is "equal to" this one.
 * <p>
 * The {@code equals} method implements an equivalence relation
 * on non-null object references:
 * <ul>
 * <li>It is <i>reflexive</i>: for any non-null reference value
 * {@code x}, {@code x.equals(x)} should return
 * {@code true}.
 * <li>It is <i>symmetric</i>: for any non-null reference values
 * {@code x} and {@code y}, {@code x.equals(y)}
 * should return {@code true} if and only if
 * {@code y.equals(x)} returns {@code true}.
 * <li>It is <i>transitive</i>: for any non-null reference values
 * {@code x}, {@code y}, and {@code z}, if
 * {@code x.equals(y)} returns {@code true} and
 * {@code y.equals(z)} returns {@code true}, then
 * {@code x.equals(z)} should return {@code true}.
 * <li>It is <i>consistent</i>: for any non-null reference values
 * {@code x} and {@code y}, multiple invocations of
 * {@code x.equals(y)} consistently return {@code true}
 * or consistently return {@code false}, provided no
 * information used in {@code equals} comparisons on the
 * objects is modified.
 * <li>For any non-null reference value {@code x},
 * {@code x.equals(null)} should return {@code false}.
 * </ul>
 * <p>
```

```
 * The {@code equals} method for class {@code Object} implements
 * the most discriminating possible equivalence relation on objects;
 * that is, for any non-null reference values {@code x} and
 * {@code y}, this method returns {@code true} if and only
 * if {@code x} and {@code y} refer to the same object
 * ({@code x == y} has the value {@code true}).
 * <p>
 * Note that it is generally necessary to override the {@code hashCode}
 * method whenever this method is overridden, so as to maintain the
 * general contract for the {@code hashCode} method, which states
 * that equal objects must have equal hash codes.
 *
 * @param obj the reference object with which to compare.
 * @return {@code true} if this object is the same as the obj
 * argument; {@code false} otherwise.
 * @see #hashCode()
 * @see HashMap
 */
@Override
public boolean equals(Object obj) {

    if (obj == null) {
        return false;
    }
    if (!(obj instanceof NulsDigestData)) {
        return false;
    }
    try {
        if (this.serialize() == null || ((NulsDigestData) obj).serialize() == null) {
            return false;
        }

        if (this.serialize().length != ((NulsDigestData) obj).serialize().length) {
            return false;
        }
    } catch (Exception e) {
        return false;
    }
    return Arrays.equals(this.getDigestBytes(), ((NulsDigestData) obj).getDigestBytes());
}

@Override
```

```
public String toString() {
    return getDigestHex();
}

/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
 * {@link HashMap}.
 * <p>
 * The general contract of {@code hashCode} is:
 * <ul>
 * <li>Whenever it is invoked on the same object more than once during
 * an execution of a Java application, the {@code hashCode} method
 * must consistently return the same integer, provided no information
 * used in {@code equals} comparisons on the object is modified.
 * This integer need not remain consistent from one execution of an
 * application to another execution of the same application.
 * <li>If two objects are equal according to the {@code equals(Object)}
 * method, then calling the {@code hashCode} method on each of
 * the two objects must produce the same integer result.
 * <li>It is <em>not</em> required that if two objects are unequal
 * according to the {@link Object#equals(Object)}
 * method, then calling the {@code hashCode} method on each of the
 * two objects must produce distinct integer results.  However, the
 * programmer should be aware that producing distinct integer results
 * for unequal objects may improve the performance of hash tables.
 * </ul>
 * <p>
 * As much as is reasonably practical, the hashCode method defined by
 * class {@code Object} does return distinct integers for distinct
 * objects. (This is typically implemented by converting the internal
 * address of the object into an integer, but this implementation
 * technique is not required by the
 * Java&trade; programming language.)
 *
 * @return a hash code value for this object.
 * @see Object#equals(Object)
 * @see System#identityHashCode
 */
@Override
public int hashCode() {
    return Arrays.hashCode(this.getDigestBytes());
```

```
    }
}

41:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\NulsSignData.java
 */
package io.nuls.kernel.model;

import io.nuls.core.tools.crypto.ECKey;
import io.nuls.core.tools.crypto.Hex;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;
import java.math.BigInteger;

/**
 * @author facjas
 */
public class NulsSignData extends BaseNulsData {

    public static byte SIGN_ALG_ECC = (short) 0;
    public static byte SIGN_ALG_DEFAULT = NulsSignData.SIGN_ALG_ECC;
    /**
     *
     */
    protected byte signAlgType;

    /**
     *
     */
    protected byte[] signBytes;

    public byte getSignAlgType() {
        return signAlgType;
    }

    public void setSignAlgType(byte signAlgType) {
        this.signAlgType = signAlgType;
```

```
    }

    public NulsSignData() {
    }

    @Override
    public int size() {
        return SerializeUtils.sizeOfBytes(signBytes) + 1;
    }

    @Override
    public void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        stream.write(signAlgType);
        stream.writeBytesWithLength(signBytes);
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        this.signAlgType = byteBuffer.readByte();
        this.signBytes = byteBuffer.readByLengthByte();
    }



    public byte[] getSignBytes() {
        return signBytes;
    }

    public void setSignBytes(byte[] signBytes) {
        this.signBytes = signBytes;
    }



    public NulsSignData sign(NulsDigestData nulsDigestData, short signAlgType, BigInteger
privkey) {
        if (signAlgType == NulsSignData.SIGN_ALG_ECC) {
            ECKey ecKey = ECKey.fromPrivate(privkey);
            byte[] signBytes = ecKey.sign(nulsDigestData.getDigestBytes(), privkey);
            NulsSignData signData = new NulsSignData();
            try {
                signData.parse(signBytes, 0);
            } catch (NulsException e) {
                Log.error(e);
```

```
            }
            return signData;
        }
        return null;
    }

    public NulsSignData sign(NulsDigestData nulsDigestData, BigInteger privkey) {
        short signAlgType = NulsSignData.SIGN_ALG_DEFAULT;
        if (signAlgType == NulsSignData.SIGN_ALG_ECC) {
            ECKey ecKey = ECKey.fromPrivate(privkey);
            return sign(nulsDigestData, signAlgType, privkey);
        }
        return null;
    }

    @Override
    public String toString() {
        try {
            return Hex.encode(serialize());
        } catch (IOException e) {
            Log.error(e);
            return super.toString();
        }
    }
}

42:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\NulsVersion.java
 */
package io.nuls.kernel.model;

/**
 * @author vivi
 */
public interface NulsVersion {

    String getVersion();

    String getArtifactId();

    String getGroupId();
}
```

```
43:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\Result.java
 */
package io.nuls.kernel.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.core.tools.json.JSONUtils;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.constant.ErrorCode;
import io.nuls.kernel.constant.KernelErrorCode;

import java.io.Serializable;

/**
 * @author vivi
 */
public class Result<T> implements Serializable {

    private boolean success;

    private String msg;

    private ErrorCode errorCode;

    private T data;


    public Result() {
        this(true, KernelErrorCode.SUCCESS, null);
    }

    public Result(boolean success) {
        this.success = success;
        this.errorCode = KernelErrorCode.SUCCESS;
    }

    public Result(boolean success, ErrorCode errorCode, T data) {
        this.success = success;
        this.errorCode = errorCode;
        this.data = data;
    }
```

```
public Result(boolean success, ErrorCode errorCode) {
    this.success = success;
    this.errorCode = errorCode;
}

public boolean isSuccess() {
    return success;
}

@JsonIgnore
public boolean isFailed() {
    return !success;
}

public Result<T> setSuccess(boolean success) {
    this.success = success;
    return this;
}

public String getMsg() {
    if (StringUtils.isBlank(msg)) {
        return errorCode.getMsg();
    }
    return msg;
}

public Result<T> setMsg(String msg) {
    this.msg = msg;
    return this;
}

public ErrorCode getErrorCode() {
    return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
}

@Override
public String toString() {
```

```
        StringBuffer buffer = new StringBuffer();
        buffer.append("result:{");
        buffer.append("\"success\": " + success + ",");
        buffer.append("\"validator\": \"" + msg + "\",");
        if (errorCode == null) {
            buffer.append("\"errorCode\": \"\",");
        } else {
            buffer.append("\"errorCode\": \"" + errorCode.getCode() + "\",");
        }
        if (data != null) {
            try {
                buffer.append("\"data\":" + JSONUtils.obj2json(data));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        buffer.append("}");
        return buffer.toString();
    }

    public static Result getFailed() {
        return getFailed(KernelErrorCode.FAILED);
    }

/*   public static Result getFailed(String msg) {
        return new Result(false, msg);
    }*/

    public static Result getSuccess() {
        return new Result(true);
    }



    public static Result getFailed(ErrorCode errorCode) {
        Result result = new Result(false, errorCode, null);
        return result;
    }

    public T getData() {
        return data;
    }
```

```
    public Result<T> setData(T data) {
        this.data = data;
        return this;
    }

    public RpcClientResult toRpcClientResult() {
        RpcClientResult rpcClientResult = new RpcClientResult();
        rpcClientResult.setSuccess(success);
        if (success) {
            rpcClientResult.setData(data);
        } else {
            ErrorData errorData = new ErrorData();
            if (StringUtils.isBlank(msg) && null != errorCode) {
                errorData.setMsg(this.errorCode.getMsg());
                errorData.setCode(this.errorCode.getCode());
            } else {
                errorData.setCode(KernelErrorCode.FAILED.getCode());
                errorData.setMsg(msg);
            }
            rpcClientResult.setData(errorData);
        }
        return rpcClientResult;
    }
}
```

44:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\model\RpcClientResult.java

```
package io.nuls.kernel.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.nuls.kernel.constant.ErrorCode;
import io.nuls.kernel.constant.KernelErrorCode;

import java.util.Map;

/**
 * @author Niels
 */
public class RpcClientResult {
```

```
private boolean success;

private Object data;

public RpcClientResult() {

}

public RpcClientResult(boolean success, ErrorData errorData) {
    this.success = success;
    this.data = errorData;
}

public RpcClientResult(boolean success, ErrorCode errorCode) {
    this.success = success;
    this.data = ErrorData.getErrorData(errorCode);
}

public RpcClientResult(boolean success, Object data) {
    this.success = success;
    this.data = data;
}

public static RpcClientResult getFailed(ErrorData errorData) {
    return new RpcClientResult(false, errorData);
}

public static RpcClientResult getFailed(ErrorCode errorCode) {
    return new RpcClientResult(false, errorCode);
}

public static RpcClientResult getFailed(String msg) {
    return getFailed(new ErrorData(KernelErrorCode.FAILED.getCode(), msg));
}

@Override
public String toString() {
    try {
        return new ObjectMapper().writeValueAsString(this);
    } catch (JsonProcessingException e) {
        return null;
    }
```

```
    }

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }

    @JsonIgnore
    public boolean isFailed() {
        return !success;
    }

    public boolean isSuccess() {
        return success;
    }

    public void setSuccess(boolean success) {
        this.success = success;
    }

    public boolean dataToBooleanValue() {
        return (boolean) ((Map) data).get("value");
    }

    public String dataToStringValue() {
        Object object = ((Map) data).get("value");
        if (null != object) {
            return (String) object;
        }
        return null;
    }
}

45:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\Transaction.java
 */
package io.nuls.kernel.model;
```

```
import io.nuls.core.tools.crypto.UnsafeByteArrayOutputStream;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.constant.TxStatusEnum;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.func.TimeService;
import io.nuls.kernel.script.SignatureUtil;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.*;

/**
 * @author Niels
 */
public abstract class Transaction<T extends TransactionLogicData> extends BaseNulsData
implements Cloneable {

    protected int type;

    protected CoinData coinData;

    protected T txData;

    protected long time;

    private byte[] transactionSignature;

    protected byte[] remark;

    protected transient NulsDigestData hash;

    protected long blockHeight = -1L;

    protected transient TxStatusEnum status = TxStatusEnum.UNCONFIRM;

    protected transient int size;
```

```
@Override
public int size() {
    int size = 0;
    size += SerializeUtils.sizeOfUint16(); // type
    size += SerializeUtils.sizeOfUint48(); // time
    size += SerializeUtils.sizeOfBytes(remark);
    size += SerializeUtils.sizeOfNulsData(txData);
    size += SerializeUtils.sizeOfNulsData(coinData);
    size += SerializeUtils.sizeOfBytes(transactionSignature);
    return size;
}

@Override
public void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
    stream.writeUint16(type);
    stream.writeUint48(time);
    stream.writeBytesWithLength(remark);
    stream.writeNulsData(txData);
    stream.writeNulsData(coinData);
    stream.writeBytesWithLength(transactionSignature);
}

public byte[] serializeForHash() throws IOException {
    ByteArrayOutputStream bos = null;
    try {
        int size = size() - SerializeUtils.sizeOfBytes(transactionSignature);

        bos = new UnsafeByteArrayOutputStream(size);
        NulsOutputStreamBuffer buffer = new NulsOutputStreamBuffer(bos);
        do {
            if (size == 0) {
                bos.write(NulsConstant.PLACE_HOLDER);
                break;
            }
            if (NulsContext.MAIN_NET_VERSION == 1) {
                buffer.writeVarInt(type);
                buffer.writeVarInt(time);
                break;
            }
            if (this.blockHeight == -1) {
                buffer.writeUint16(type);
```

```
                buffer.writeUint48(time);
                break;
            }
            if (NulsContext.CHANGE_HASH_SERIALIZE_HEIGHT != null && this.blockHeight >=
NulsContext.CHANGE_HASH_SERIALIZE_HEIGHT) {
                buffer.writeUint16(type);
                buffer.writeUint48(time);
            } else {
                buffer.writeVarInt(type);
                buffer.writeVarInt(time);
            }
        } while (false);
        if (size > 0) {
            buffer.writeBytesWithLength(remark);
            buffer.writeNulsData(txData);
            buffer.writeNulsData(coinData);
        }
        return bos.toByteArray();
    } finally {
        if (bos != null) {
            try {
                bos.close();
            } catch (IOException e) {
                throw e;
            }
        }
    }
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    type = byteBuffer.readUint16();
    time = byteBuffer.readUint48();
    this.remark = byteBuffer.readByLengthByte();
    txData = this.parseTxData(byteBuffer);
    this.coinData = byteBuffer.readNulsData(new CoinData());
    /*try {
        hash = NulsDigestData.calcDigestData(this.serializeForHash());
    } catch (IOException e) {
        Log.error(e);
    }*/
    transactionSignature = byteBuffer.readByLengthByte();
```

```
    }

    //
//    /**
//     *
//     * Is a system to produce trading (packaged node generation, for the piece reward settlement, CARDS punishment),
//     * trading in the validation of this kind of new type block size is not taken into account, the types of transactions do not need poundage
//     */
    public boolean isSystemTx() {
        return false;
    }

    //
//    /**
//     * -1UTXOUTXO
//     * If it's an unlocking transaction, this type of transaction costs the UTXO with a lock time of -1 and generates a new UTXO
//     */
    public boolean isUnlockTx() {
        return false;
    }

    //
//    /**
//     *
//     * If the deal need to verify the signature in the book, all transactions system and some special deal,
//     * no need to install the ordinary transaction way to verify the signature, will provide additional validation logic.
//     */
    public boolean needVerifySignature() {
        return true;
    }

    protected abstract T parseTxData(NulsByteBuffer byteBuffer) throws NulsException;

    public Transaction(int type) {
        this.time = TimeService.currentTimeMillis();
        this.type = type;
    }
```

```
public long getTime() {
    return time;
}

public void setTime(long time) {
    this.time = time;
}

public void setType(int type) {
    this.type = type;
}

public int getType() {
    return type;
}

public byte[] getRemark() {
    return remark;
}

public void setRemark(byte[] remark) {
    this.remark = remark;
}

public NulsDigestData getHash() {
    if (hash == null) {
        try {
            hash = NulsDigestData.calcDigestData(serializeForHash());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return hash;
}

public void setHash(NulsDigestData hash) {
    this.hash = hash;
}

public byte[] getTransactionSignature() {
    return transactionSignature;
```

```
}

public void setTransactionSignature(byte[] transactionSignature) {
    this.transactionSignature = transactionSignature;
}

public T getTxData() {
    return txData;
}

public void setTxData(T txData) {
    this.txData = txData;
}

public long getBlockHeight() {
    return blockHeight;
}

public void setBlockHeight(long blockHeight) {
    this.blockHeight = blockHeight;
}

public TxStatusEnum getStatus() {
    return status;
}

public void setStatus(TxStatusEnum status) {
    this.status = status;
}

public CoinData getCoinData() {
    return coinData;
}

public void setCoinData(CoinData coinData) {
    this.coinData = coinData;
}

public int getSize() {
    if (size == 0) {
        size = size();
    }
```

```
    return size;
}

public void setSize(int size) {
    this.size = size;
}

public Na getFee() {
    if (isSystemTx()) {
        return Na.ZERO;
    }
    Na fee = Na.ZERO;
    if (null != coinData) {
        fee = coinData.getFee();
    }
    return fee;
}

public List<byte[]> getAllRelativeAddress() {
    Set<byte[]> addresses = new HashSet<>();

    if (coinData != null) {
        Set<byte[]> coinAddressSet = coinData.getAddresses();
        if (null != coinAddressSet) {
            addresses.addAll(coinAddressSet);
        }
    }
    if (txData != null) {
        Set<byte[]> txAddressSet = txData.getAddresses();
        if (null != txAddressSet) {
            addresses.addAll(txAddressSet);
        }
    }
    if (this.transactionSignature != null) {
        try {
            Set<String> signAddresss = SignatureUtil.getAddressFromTX(this);
            for (String signAddr : signAddresss) {
                boolean hasExist = false;
                for (byte[] addr : addresses) {
                    if (Arrays.equals(AddressTool.getAddress(signAddr), addr)) {
                        hasExist = true;
                        break;
```

```
                    }
                }
                if (!hasExist) {
                    addresses.add(AddressTool.getAddress(signAddr));
                }
            }
        } catch (NulsException e) {
            Log.error(e);
        }
    }
    return new ArrayList<>(addresses);
}

public abstract String getInfo(byte[] address);

@Override
public String toString() {
    return "Transaction{" +
            "type=" + type +
            ", coinData=" + coinData +
            ", txData=" + txData +
            ", time=" + time +
            ", hash=" + hash +
            ", blockHeight=" + blockHeight +
            ", status=" + status +
            ", size=" + size +
            '}';
}
}

46:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\model\TransactionLogicData.java
 */

package io.nuls.kernel.model;

import java.util.Set;

/**
 * author Facjas
 * date 2018/5/10.
 */
```

```
public abstract class TransactionLogicData extends BaseNulsData {
    public abstract Set<byte[]> getAddresses();
}

47:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\module\BaseModuleBootstrap.java
 */
package io.nuls.kernel.module;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.module.manager.ModuleManager;

/**
 * @author Niels
 */
public abstract class BaseModuleBootstrap {

    private final short moduleId;

    private String moduleName;

    private ModuleStatusEnum status;

    public BaseModuleBootstrap(short moduleId) {
        this.moduleId = moduleId;
        this.status = ModuleStatusEnum.UNINITIALIZED;
    }

    public abstract void init() throws Exception;

    /**
     * start the module
     */
    public abstract void start();

    /**
     * stop the module
     */
    public abstract void shutdown();
```

```
    /**
     * destroy the module
     */
    public abstract void destroy();

    //    /**
//     * get all info of the module
//     */
    public abstract String getInfo();

    //    /**
//     * get the status of the module
//     */
    public final ModuleStatusEnum getStatus() {
        return this.status;
    }

    public final String getModuleName() {
        return moduleName;
    }

    public void setStatus(ModuleStatusEnum status) {
        Log.info("Status change(" + this.moduleName + "):" + this.status + "-->" + status);
        this.status = status;
    }

    protected final String getModuleCfgProperty(String section, String property) {
        try {
            return NulsConfig.MODULES_CONFIG.getCfgValue(section, property);
        } catch (Exception e) {
            Log.error(e);
            return null;
        }
    }

//    protected final void registerTransaction(int txType, Class<? extends Transaction> txClass,
Class<? extends TransactionService> txServiceClass) {
//        this.registerService(txServiceClass);
//        TransactionManager.putTx(txType, txClass, txServiceClass);
//    }

    public Class<? extends BaseModuleBootstrap> getModuleClass() {
```

```
        return this.getClass();
    }

    public short getModuleId() {
        return moduleId;
    }

    public void setModuleName(String moduleName) {
        this.moduleName = moduleName;
    }

    protected void waitForDependencyInited(short... moduleIds) {
        ModuleManager mm = ModuleManager.getInstance();
        String result = checkItInited(mm, moduleIds);
        while (result != null) {
            try {
                Thread.sleep(100L);
            } catch (InterruptedException e) {
                Log.error(e);
            }
            result = checkItInited(mm, moduleIds);
        }
    }

    private String checkItInited(ModuleManager mm, short[] moduleIds) {
        for (short id : moduleIds) {
            BaseModuleBootstrap module = mm.getModule(id);
            if (null == module) {
                return id + "";
            }
            if (null == module || module.getStatus() == ModuleStatusEnum.UNINITIALIZED ||
module.getStatus() == ModuleStatusEnum.INITIALIZING || module.getStatus() ==
ModuleStatusEnum.EXCEPTION) {
                return module.getModuleName();
            }
        }
        return null;
    }


    protected void waitForDependencyRunning(short... moduleIds) {
        ModuleManager mm = ModuleManager.getInstance();
```

```
        String result = checkItRunning(mm, moduleIds);
        while (result != null) {
            try {
                Thread.sleep(100L);
            } catch (InterruptedException e) {
                Log.error(e);
            }
            result = checkItRunning(mm, moduleIds);
        }
    }

    private String checkItRunning(ModuleManager mm, short[] moduleIds) {
        for (short id : moduleIds) {
            BaseModuleBootstrap module = mm.getModule(id);
            if (null == module) {
                return "null";
            }
            if (module.getStatus() != ModuleStatusEnum.RUNNING) {
                return module.getModuleName();
            }
        }
        return null;
    }
}

48:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\module\manager\ModuleManager.java
 */
package io.nuls.kernel.module.manager;


import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.module.thread.ModuleProcess;
import io.nuls.kernel.module.thread.ModuleProcessFactory;
import io.nuls.kernel.module.thread.ModuleRunner;

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

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author Niels
 */
public class ModuleManager {

    private Map<String, String> modulesCfg;

    private final Map<Short, ModuleProcess> PROCCESS_MAP = new ConcurrentHashMap<>();

    private ModuleProcessFactory factory = new ModuleProcessFactory();

    private static final Map<Short, BaseModuleBootstrap> MODULE_MAP = new
ConcurrentHashMap<>();

    private static final ModuleManager MANAGER = new ModuleManager();

    private ModuleManager() {
    }

    public static ModuleManager getInstance() {
        return MANAGER;
    }

    public BaseModuleBootstrap getModule(short moduleId) {
        return MODULE_MAP.get(moduleId);
    }

    public Short getModuleId(Class<? extends BaseModuleBootstrap> moduleClass) {
        if (null == moduleClass) {
            return null;
        }

        for (BaseModuleBootstrap module : MODULE_MAP.values()) {
            if (moduleClass.equals(module.getClass()) ||
isImplements(module.getClass().getSuperclass(), moduleClass)) {
                return module.getModuleId();
            }
        }
        try {
```

```
            Thread.sleep(100L);
            Log.warn("wait for the module init:" + moduleClass);
        } catch (InterruptedException e) {
            Log.error(e);
        }
        return this.getModuleId(moduleClass);
    }

    private boolean isImplements(Class superClass, Class<? extends BaseModuleBootstrap>
moduleClass) {
        boolean result = moduleClass.equals(superClass);
        if (result) {
            return true;
        }
        if (Object.class.equals(superClass.getSuperclass())) {
            return false;
        }
        return isImplements(superClass.getSuperclass(), moduleClass);
    }

    public void regModule(BaseModuleBootstrap module) {
        short moduleId = module.getModuleId();
        if (MODULE_MAP.keySet().contains(moduleId)) {
            throw new NulsRuntimeException(KernelErrorCode.THREAD_REPETITION);
        }
        MODULE_MAP.put(moduleId, module);
    }

    public void remModule(short moduleId) {
        MODULE_MAP.remove(moduleId);
    }


    public void stopModule(short moduleId) {
        BaseModuleBootstrap module = MODULE_MAP.get(moduleId);
        if (null == module) {
            return;
        }
        module.shutdown();
        ModuleProcess process = PROCCESS_MAP.get(moduleId);
        if (null != process && !process.isInterrupted()) {
            process.interrupt();
```

```
    }
}

public void destroyModule(short moduleId) {
    BaseModuleBootstrap module = MODULE_MAP.get(moduleId);
    if (null == module) {
        return;
    }
    module.setStatus(ModuleStatusEnum.DESTROYING);
    try {
        if (module.getStatus() != ModuleStatusEnum.STOPED) {
            stopModule(moduleId);
        }
        module.destroy();
        remModule(module.getModuleId());
        removeProcess(module.getModuleId());
        module.setStatus(ModuleStatusEnum.DESTROYED);
    } catch (Exception e) {
        module.setStatus(ModuleStatusEnum.EXCEPTION);
    }

}

public String getInfo() {
    StringBuilder str = new StringBuilder("Message:");
    for (BaseModuleBootstrap module : MODULE_MAP.values()) {
        str.append("\nModule:");
        str.append(module.getModuleName());
        str.append("");
        str.append("id(");
        str.append(module.getModuleId());
        str.append("),");
        str.append("status:");
        str.append(module.getStatus());
        str.append("\nINFO:");
        str.append(module.getInfo());
    }
    return str.toString();
}

public ModuleStatusEnum getModuleState(short moduleId) {
    BaseModuleBootstrap module = MODULE_MAP.get(moduleId);
```

```
    if (null == module) {
        return ModuleStatusEnum.NOT_FOUND;
    }
    if (ModuleStatusEnum.RUNNING == module.getStatus()) {
        Thread.State state = getProcessState(moduleId);
        if (state == Thread.State.TERMINATED) {
            module.setStatus(ModuleStatusEnum.EXCEPTION);
        }
    }
    return module.getStatus();
}

public void startModule(String key, String moduleClass) {
    if (null == moduleClass) {
        return;
    }
    try {
        ModuleRunner runner = new ModuleRunner(key, moduleClass);
        ModuleProcess moduleProcess = factory.newThread(runner);
        moduleProcess.start();
    } catch (Exception e) {
        Log.error(e);
    }
}

private Thread.State getProcessState(short moduleId) {
    ModuleProcess process = PROCCESS_MAP.get(moduleId);
    if (null != process) {
        return process.getState();
    }
    return null;
}

public List<ModuleProcess> getProcessList() {
    return new ArrayList<>(PROCCESS_MAP.values());
}

private void removeProcess(short moduleId) {
    this.destroyModule(moduleId);
    PROCCESS_MAP.remove(moduleId);
}
```

```
    public Map<String, String> getModulesCfg() {
        return modulesCfg;
    }

    public void setModulesCfg(Map<String, String> modulesCfg) {
        this.modulesCfg = modulesCfg;
    }

    public List<BaseModuleBootstrap> getModuleList() {
        return new ArrayList<>(MODULE_MAP.values());
    }
}
```

49:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\module\manager\ServiceManager.java

```
 */
package io.nuls.kernel.module.manager;

import io.nuls.core.tools.param.AssertUtil;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.module.service.ModuleService;

import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author Niels
 */
public class ServiceManager {
    private static final ServiceManager INSTANCE = new ServiceManager();
    private static final Map<Short, Set<Class>> MODULE_INTF_MAP = new ConcurrentHashMap<>();
    private static final Map<Class, Short> MODULE_ID_MAP = new ConcurrentHashMap<>();

    private ServiceManager() {
    }
```

```
public static final ServiceManager getInstance() {
    return INSTANCE;
}

public void regService(short moduleId, Class serviceClass) throws NulsException {
    AssertUtil.canNotEmpty(serviceClass, KernelErrorCode.NULL_PARAMETER.getMsg());
    if (MODULE_ID_MAP.containsKey(serviceClass)) {
        return;
    }
    SpringLiteContext.putBean(serviceClass);
    Set<Class> set = MODULE_INTF_MAP.get(moduleId);
    if (null == set) {
        set = new HashSet<>();
    }
    set.add(serviceClass);
    MODULE_INTF_MAP.put(moduleId, set);
    MODULE_ID_MAP.put(serviceClass, moduleId);
}


public void removeService(short moduleId, Class clazz) {
    MODULE_ID_MAP.remove(clazz);
    Set<Class> set = MODULE_INTF_MAP.get(moduleId);
    if (null != set) {
        set.remove(clazz);
        MODULE_INTF_MAP.put(moduleId, set);
    }
    SpringLiteContext.removeBean(clazz);
}

public void removeService(short moduleId) {
    Set<Class> set = MODULE_INTF_MAP.get(moduleId);
    if (null == set) {
        return;
    }
    for (Class clazz : set) {
        removeService(moduleId, clazz);
    }
}

public BaseModuleBootstrap getModule(Class clazz) {
    Short moduleId = MODULE_ID_MAP.get(clazz);
```

```
        if (moduleId == null) {
            return null;
        }
        return ModuleService.getInstance().getModule(moduleId);
    }
}

50:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\module\service\ModuleService.java
 */
package io.nuls.kernel.module.service;


import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.module.manager.ModuleManager;

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

/**
 * @author Niels
 */
public class ModuleService {

    private ModuleManager moduleManager = ModuleManager.getInstance();

    private static final ModuleService INSTANCE = new ModuleService();

    private ModuleService() {
    }

    public static ModuleService getInstance() {
        return INSTANCE;
    }

    public Short getModuleId(Class<? extends BaseModuleBootstrap> clazz) {
        return moduleManager.getModuleId(clazz);
    }
```

```
    public void startModule(String key, String moduleClass) throws IllegalAccessException,
InstantiationException, ClassNotFoundException {
        moduleManager.startModule(key, moduleClass);
    }

    public ModuleStatusEnum getModuleState(short moduleId) {
        return moduleManager.getModuleState(moduleId);
    }

    public void shutdown(short moduleId) {
        moduleManager.stopModule(moduleId);
    }

    public void destroy(short moduleId) {
        moduleManager.destroyModule(moduleId);
    }

    public void startModules(Map<String, String> bootstrapClasses) {
        moduleManager.setModulesCfg(bootstrapClasses);
        List<String> keyList = new ArrayList<>(bootstrapClasses.keySet());
        for (String key : keyList) {
            try {
                startModule(key, bootstrapClasses.get(key));
            } catch (Exception e) {
                throw new NulsRuntimeException(e);
            }
        }
    }

    public BaseModuleBootstrap getModule(short moduleId) {
        return moduleManager.getModule(moduleId);
    }

    public List<BaseModuleBootstrap> getModuleList() {
        return moduleManager.getModuleList();
    }
}

51:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\module\thread\ModuleProcess.java
 */
package io.nuls.kernel.module.thread;
```

```
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.func.TimeService;
import io.nuls.kernel.thread.BaseThread;

/**
 * @author Niels
 */
public class ModuleProcess extends BaseThread {
    private final ModuleRunner runner;
    private long startTime;
    private boolean running = true;

    public ModuleProcess(ModuleRunner target) {
        super(target);
        this.runner = target;
    }

    @Override
    public void beforeStart() {
        this.startTime = TimeService.currentTimeMillis();
    }

    @Override
    protected void afterStart() {

    }

    @Override
    protected void runException(Exception e) {
        Log.error(e);
        runner.getModule().setStatus(ModuleStatusEnum.EXCEPTION);
        running = false;
    }

    @Override
    protected void afterRun() {
        while (running && ModuleStatusEnum.RUNNING.equals(runner.getModule().getStatus())) {
            try {
                Thread.sleep(10000L);
            } catch (InterruptedException e) {
```

```
                Log.error(e);
            }
        }
    }

    @Override
    protected void beforeRun() {
    }

    @Override
    protected void afterInterrupt() {
        runner.getModule().setStatus(ModuleStatusEnum.STOPED);

    }

    @Override
    protected void beforeInterrupt() {
        runner.getModule().setStatus(ModuleStatusEnum.STOPPING);
        running = false;
    }

    public long getStartTime() {
        return startTime;
    }

    public ModuleRunner getRunner() {
        return runner;
    }
}
```

52:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\module\thread\ModuleProcessFactory.java

```
 */
package io.nuls.kernel.module.thread;

import java.util.concurrent.ThreadFactory;

/**
 * @author Niels
 */
public class ModuleProcessFactory implements ThreadFactory {
```

```
    private static final String POOL_NAME = "Process";

    @Override
    public ModuleProcess newThread(Runnable r) {
        if (null == r) {
            throw new RuntimeException("runnable cannot be null!");
        }
        if (!(r instanceof ModuleRunner)) {
            throw new RuntimeException("unkown runnable!");
        }
        ModuleProcess process = new ModuleProcess((ModuleRunner) r);
        process.setModuleId((short) 0);
        process.setName(POOL_NAME + "-" + ((ModuleRunner) r).getModuleKey());
        process.setPoolName(POOL_NAME);
        process.setDaemon(false);
        return process;
    }
}

53:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\module\thread\ModuleRunner.java
 */
package io.nuls.kernel.module.thread;

import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.constant.ModuleStatusEnum;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.module.BaseModuleBootstrap;
import io.nuls.kernel.module.manager.ModuleManager;

/**
 * @author Niels
 */
public class ModuleRunner implements Runnable {

    private final String moduleKey;
    private final String moduleClass;
    private BaseModuleBootstrap module;

    public ModuleRunner(String key, String moduleClass) {
        this.moduleKey = key;
```

```
        this.moduleClass = moduleClass;
    }

    @Override
    public void run() {
        try {
            module = this.loadModule();
            module.setStatus(ModuleStatusEnum.INITIALIZING);
            module.init();
            module.setStatus(ModuleStatusEnum.INITIALIZED);
            module.setStatus(ModuleStatusEnum.STARTING);
            module.start();
            module.setStatus(ModuleStatusEnum.RUNNING);
        } catch (ClassNotFoundException e) {
//            module.setStatus(ModuleStatusEnum.EXCEPTION);
            Log.error(e);
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            module.setStatus(ModuleStatusEnum.EXCEPTION);
            Log.error(e);
            throw new RuntimeException(e);
        } catch (InstantiationException e) {
            module.setStatus(ModuleStatusEnum.EXCEPTION);
            Log.error(e);
            throw new RuntimeException(e);
        } catch (NulsRuntimeException e) {
            module.setStatus(ModuleStatusEnum.EXCEPTION);
            Log.error(e);
            throw e;
        } catch (Exception e) {
            module.setStatus(ModuleStatusEnum.EXCEPTION);
            Log.error(e);
            System.exit(-1);
        }
    }

    private BaseModuleBootstrap loadModule() throws ClassNotFoundException,
IllegalAccessException, InstantiationException {
        BaseModuleBootstrap module = null;
        do {
            if (StringUtils.isBlank(moduleClass)) {
                Log.warn("module cannot start:" + moduleClass);
```

```
                break;
            }
            Class clazz = Class.forName(moduleClass);
            module = (BaseModuleBootstrap) clazz.newInstance();
            module.setModuleName(this.moduleKey);
            Log.debug("load module:" + module.getInfo());
        } while (false);
        ModuleManager.getInstance().regModule(module);

        return module;
    }

    public String getModuleKey() {
        return moduleKey;
    }

    public BaseModuleBootstrap getModule() {
        return module;
    }
}

54:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\processor\CommandProcessor.java
 */

package io.nuls.kernel.processor;

import io.nuls.kernel.model.CommandResult;

/**
 * RPC
 */
public interface CommandProcessor {

    String getCommand();

    String getHelp();

    String getCommandDescription();

    boolean argsValidate(String[] args);
```

```
    CommandResult execute(String[] args);
}

55:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\processor\ConflictDetectProcessor.java
 */

package io.nuls.kernel.processor;

import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.validate.ValidateResult;

import java.util.List;

/**
 * @author: Niels Wang
 */
public interface ConflictDetectProcessor {

    /**
     *
     *
     * <p>
     * Conflict detection, which detects conflicting transactions in the incoming transaction list, returns failure,
     * indicating the cause of failure and the transaction that should be discarded.
     * This method does not check the double flower conflict, the double flower is realized by the accounting interface.
     *
     * @param txList /A list of transactions to be checked.
     * @return successResultdatamsg
     * Operation result: success returns successResult. When failure, data returns the discard list, and MSG returns the cause of conflict.
     */
    ValidateResult conflictDetect(List<Transaction> txList);
}

56:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\processor\TransactionProcessor.java
 */
package io.nuls.kernel.processor;
```

```
import io.nuls.kernel.model.Result;
import io.nuls.kernel.model.Transaction;

/**
 *
 * {@link io.nuls.kernel.lite.annotation.Service}
 * Transaction processors, each transaction needs to implement its own transaction cmd,
 * deal with the business of the transaction itself, and different business methods are invoked by
different life cycles.
 * Implement this interface class, you need to add {@link io.nuls.kernel.lite.annotation.Service}
annotation
 *
 * @author Niels
 */
public interface TransactionProcessor<T extends Transaction> extends ConflictDetectProcessor {

    //    /**
//     *
//     * This method is called when the transaction rolls back.
//     *
//     * @param tx            The transaction to roll back.
//     * @param secondaryData Secondary data, depending on the business needs to be passed.
//     */
    Result onRollback(T tx, Object secondaryData);

    //    /**
//     *
//     * This method is called when the transaction save.
//     *
//     * @param tx            The transaction to save;
//     * @param secondaryData Secondary data, depending on the business needs to be passed.
//     */
    Result onCommit(T tx, Object secondaryData);


}

57:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\BlockSignature.java
 */

package io.nuls.kernel.script;
```

```
import io.nuls.core.tools.crypto.ECKey;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BaseNulsData;
import io.nuls.kernel.model.NulsDigestData;
import io.nuls.kernel.model.NulsSignData;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;
import io.nuls.kernel.validate.ValidateResult;

import java.io.IOException;

public class BlockSignature extends BaseNulsData {
    private NulsSignData signData;
    private byte[] publicKey;

    /**
     * serialize important field
     */
    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        stream.write(publicKey.length);
        stream.write(publicKey);
        stream.writeNulsData(signData);
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        int length = byteBuffer.readByte();
        this.publicKey = byteBuffer.readBytes(length);
        this.signData = new NulsSignData();
        this.signData.parse(byteBuffer);
    }

    @Override
    public int size() {
        int size = 1 + publicKey.length;
        size += SerializeUtils.sizeOfNulsData(signData);
        return size;
    }
```

```
    public ValidateResult verifySignature(NulsDigestData digestData) {
        boolean b = ECKey.verify(digestData.getDigestBytes(), signData.getSignBytes(), publicKey);
        if (b) {
            return ValidateResult.getSuccessResult();
        } else {
            return ValidateResult.getFailedResult(this.getClass().getName(),
KernelErrorCode.SIGNATURE_ERROR);
        }
    }

    public byte[] getPublicKey() {
        return publicKey;
    }

    public void setPublicKey(byte[] publicKey) {
        this.publicKey = publicKey;
    }

    public NulsSignData getSignData() {
        return signData;
    }

    public void setSignData(NulsSignData signData) {
        this.signData = signData;
    }
}

58:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\P2PHKSignature.java
 */

package io.nuls.kernel.script;

import com.google.common.primitives.UnsignedBytes;
import io.nuls.core.tools.crypto.ECKey;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BaseNulsData;
import io.nuls.kernel.model.NulsDigestData;
import io.nuls.kernel.model.NulsSignData;
```

```
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;
import io.nuls.kernel.validate.ValidateResult;

import java.io.IOException;
import java.util.Comparator;

public class P2PHKSignature extends BaseNulsData {

    public static final int SERIALIZE_LENGTH = 110;

    private NulsSignData signData;

    private byte[] publicKey;

    public P2PHKSignature() {
    }

    public P2PHKSignature(byte[] signBytes, byte[] publicKey) {
        this.signData = new NulsSignData();
        try {
            this.signData.parse(signBytes, 0);
        } catch (NulsException e) {
            Log.error(e);
        }
        this.publicKey = publicKey;
    }

    public P2PHKSignature(NulsSignData signData, byte[] publicKey) {
        this.signData = signData;
        this.publicKey = publicKey;
    }

    public NulsSignData getSignData() {
        return signData;
    }

    public void setSignData(NulsSignData signData) {
        this.signData = signData;
    }
```

```
    public byte[] getPublicKey() {
        return publicKey;
    }

    public void setPublicKey(byte[] publicKey) {
        this.publicKey = publicKey;
    }

    public ValidateResult verifySign(NulsDigestData digestData) {
        boolean b = ECKey.verify(digestData.getDigestBytes(), signData.getSignBytes(),
this.getPublicKey());
        if (b) {
            return ValidateResult.getSuccessResult();
        } else {
            return ValidateResult.getFailedResult(this.getClass().getName(),
KernelErrorCode.SIGNATURE_ERROR);
        }
    }

    public byte[] getSignerHash160() {
        return SerializeUtils.sha256hash160(getPublicKey());
    }

    public static P2PHKSignature createFromBytes(byte[] bytes) throws NulsException {
        P2PHKSignature sig = new P2PHKSignature();
        sig.parse(bytes, 0);
        return sig;
    }

    public byte[] getBytes() {
        try {
            return this.serialize();
        } catch (IOException e) {
            Log.error(e);
            return null;
        }
    }

    /**
     * serialize important field
     */
    @Override
```

```
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        stream.write(publicKey.length);
        stream.write(publicKey);
        stream.writeNulsData(signData);
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        int length = byteBuffer.readByte();
        this.publicKey = byteBuffer.readBytes(length);
        this.signData = new NulsSignData();
        this.signData.parse(byteBuffer);
    }

    @Override
    public int size() {
        int size = 1 + publicKey.length;
        size += SerializeUtils.sizeOfNulsData(signData);
        return size;
    }

    public ValidateResult verifySignature(NulsDigestData digestData) {
        boolean b = ECKey.verify(digestData.getDigestBytes(), signData.getSignBytes(), publicKey);
        if (b) {
            return ValidateResult.getSuccessResult();
        } else {
            return ValidateResult.getFailedResult(this.getClass().getName(),
KernelErrorCode.SIGNATURE_ERROR);
        }
    }

    public static final Comparator<P2PHKSignature> PUBKEY_COMPARATOR = new
Comparator<P2PHKSignature>() {
        private Comparator<byte[]> comparator = UnsignedBytes.lexicographicalComparator();

        @Override
        public int compare(P2PHKSignature k1, P2PHKSignature k2) {
            return comparator.compare(k1.getPublicKey(), k2.getPublicKey());
        }
    };
}
```

59:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\script\Script.java
```
import io.nuls.core.tools.crypto.Sha256Hash;
import io.nuls.core.tools.crypto.UnsafeByteArrayOutputStream;
import io.nuls.kernel.model.Address;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.SerializeUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.spongycastle.crypto.digests.RIPEMD160Digest;

import javax.annotation.Nullable;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.*;

import static com.google.common.base.Preconditions.*;
import static io.nuls.kernel.script.ScriptOpCodes.*;


// TODO: Redesign this entire API to be more type safe and organised.

/**
 * <p>Programs embedded inside transactions that control redemption of payments.</p>
 *
 * <p>Bitcoin transactions don't specify what they do directly. Instead <a
href="https://en.bitcoin.it/wiki/Script">a
 * small binary stack language</a> is used to define programs that when evaluated return whether
the transaction
 * "accepts" or rejects the other transactions connected to it.</p>
 *
 * <p>In SPV mode, scripts are not run, because that would require all transactions to be available
and lightweight
 * clients don't have that data. In full mode, this class is used to run the interpreted language. It
also has
 * static methods for building scripts.</p>
 */
```

```
public class Script {

    /**
     * Enumeration to encapsulate the type of this script.
     */
    public enum ScriptType {
        // Do NOT change the ordering of the following definitions because their ordinals are stored in databases.
        NO_TYPE,
        P2PKH,
        PUB_KEY,
        P2SH
    }

    /**
     * Flags to pass to {@link Script#correctlySpends(Transaction, long, Script, Set)}.
     * Note currently only P2SH, DERSIG and NULLDUMMY are actually supported.
     */
    public enum VerifyFlag {
        P2SH, // Enable BIP16-style subscript evaluation.
        STRICTENC, // Passing a non-strict-DER signature or one with undefined hashtype to a checksig operation causes script failure.
        DERSIG, // Passing a non-strict-DER signature to a checksig operation causes script failure (softfork safe, BIP66 rule 1)
        LOW_S, // Passing a non-strict-DER signature or one with S > order/2 to a checksig operation causes script failure
        NULLDUMMY, // Verify dummy stack item consumed by CHECKMULTISIG is of zero-length.
        SIGPUSHONLY, // Using a non-push operator in the scriptSig causes script failure (softfork safe, BIP62 rule 2).
        MINIMALDATA, // Require minimal encodings for all push operations
        DISCOURAGE_UPGRADABLE_NOPS, // Discourage use of NOPs reserved for upgrades (NOP1-10)
        CLEANSTACK, // Require that only a single stack element remains after evaluation.
        CHECKLOCKTIMEVERIFY // Enable CHECKLOCKTIMEVERIFY operation
    }

    public static final EnumSet<VerifyFlag> ALL_VERIFY_FLAGS = EnumSet.allOf(VerifyFlag.class);

    private static final Logger log = LoggerFactory.getLogger(Script.class);
    public static final long MAX_SCRIPT_ELEMENT_SIZE = 520;  // bytes
    public static final int SIG_SIZE = 75;
```

```
    /**
     * Max number of sigops allowed in a standard p2sh redeem script
     */
    public static final int MAX_P2SH_SIGOPS = 15;

    // The program is a set of chunks where each element is either [opcode] or [data, data, data ...]
    protected List<ScriptChunk> chunks;
    // Unfortunately, scripts are not ever re-serialized or canonicalized when used in signature hashing. Thus we
    // must preserve the exact bytes that we read off the wire, along with the parsed form.
    protected byte[] program;

    // Creation time of the associated keys in seconds since the epoch.
    private long creationTimeSeconds;

    /**
     * Creates an empty script that serializes to nothing.
     */
    private Script() {
        chunks = Lists.newArrayList();
    }

    // Used from ScriptBuilder.
    Script(List<ScriptChunk> chunks) {
        this.chunks = Collections.unmodifiableList(new ArrayList<ScriptChunk>(chunks));
        creationTimeSeconds = System.currentTimeMillis() / 1000;
    }

    /**
     * Construct a Script that copies and wraps the programBytes array. The array is parsed and checked for syntactic
     * validity.
     *
     * @param programBytes Array of program bytes from a transaction.
     */
    public Script(byte[] programBytes) throws ScriptException {
        program = programBytes;
        parse(programBytes);
        creationTimeSeconds = 0;
    }

    public Script(byte[] programBytes, long creationTimeSeconds) throws ScriptException {
```

```
    program = programBytes;
    parse(programBytes);
    this.creationTimeSeconds = creationTimeSeconds;
}

public long getCreationTimeSeconds() {
    return creationTimeSeconds;
}

public void setCreationTimeSeconds(long creationTimeSeconds) {
    this.creationTimeSeconds = creationTimeSeconds;
}

/**
 * Returns the program opcodes as a string, for example "[1234] DUP HASH160"
 */
@Override
public String toString() {
    return SerializeUtils.join(chunks);
}

/**
 * Returns the serialized program as a newly created byte array.
 */
public byte[] getProgram() {
    try {
        // Don't round-trip as Bitcoin Core doesn't and it would introduce a mismatch.
        if (program != null) {
            return Arrays.copyOf(program, program.length);
        }
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        for (ScriptChunk chunk : chunks) {
            chunk.write(bos);
        }
        program = bos.toByteArray();
        return program;
    } catch (IOException e) {
        throw new RuntimeException(e);  // Cannot happen.
    }
}

/**
```

```
     * Returns an immutable list of the scripts parsed form. Each chunk is either an opcode or data element.
     */
    public List<ScriptChunk> getChunks() {
        return Collections.unmodifiableList(chunks);
    }

    private static final ScriptChunk[] STANDARD_TRANSACTION_SCRIPT_CHUNKS = {
            new ScriptChunk(ScriptOpCodes.OP_DUP, null, 0),
            new ScriptChunk(ScriptOpCodes.OP_HASH160, null, 1),
            new ScriptChunk(ScriptOpCodes.OP_EQUALVERIFY, null, 23),
            new ScriptChunk(ScriptOpCodes.OP_CHECKSIG, null, 24),
    };

    /**
     * <p>To run a script, first we parse it which breaks it up into chunks representing pushes of data or logical
     * opcodes. Then we can run the parsed chunks.</p>
     *
     * <p>The reason for this split, instead of just interpreting directly, is to make it easier
     * to reach into a programs structure and pull out bits of data without having to run it.
     * This is necessary to render the to/from addresses of transactions in a user interface.
     * Bitcoin Core does something similar.</p>
     * <p>
     *
     */
    private void parse(byte[] program) throws ScriptException {
        chunks = new ArrayList<ScriptChunk>(5);   // Common size.
        ByteArrayInputStream bis = new ByteArrayInputStream(program);
        int initialSize = bis.available();
        while (bis.available() > 0) {
            int startLocationInProgram = initialSize - bis.available();
            int opcode = bis.read();

            long dataToRead = -1;
            if (opcode >= 0 && opcode < OP_PUSHDATA1) {
                // Read some bytes of data, where how many is the opcode value itself.
                dataToRead = opcode;
            } else if (opcode == OP_PUSHDATA1) {
                if (bis.available() < 1) {
                    throw new ScriptException("Unexpected end of script");
                }
```

```
                dataToRead = bis.read();
            } else if (opcode == OP_PUSHDATA2) {
                // Read a short, then read that many bytes of data.
                if (bis.available() < 2) {
                    throw new ScriptException("Unexpected end of script");
                }
                dataToRead = bis.read() | (bis.read() << 8);
            } else if (opcode == OP_PUSHDATA4) {
                // Read a uint32, then read that many bytes of data.
                // Though this is allowed, because its value cannot be > 520, it should never actually be
used
                if (bis.available() < 4) {
                    throw new ScriptException("Unexpected end of script");
                }
                dataToRead = ((long) bis.read()) | (((long) bis.read()) << 8) | (((long) bis.read()) << 16) |
(((long) bis.read()) << 24);
            }

            ScriptChunk chunk;
            if (dataToRead == -1) {
                chunk = new ScriptChunk(opcode, null, startLocationInProgram);
            } else {
                if (dataToRead > bis.available()) {
                    throw new ScriptException("Push of data element that is larger than remaining data");
                }
                byte[] data = new byte[(int) dataToRead];
                checkState(dataToRead == 0 || bis.read(data, 0, (int) dataToRead) == dataToRead);
                chunk = new ScriptChunk(opcode, data, startLocationInProgram);
            }
            // Save some memory by eliminating redundant copies of the same chunk objects.
            for (ScriptChunk c : STANDARD_TRANSACTION_SCRIPT_CHUNKS) {
                if (c.equals(chunk)) {
                    chunk = c;
                }
            }
            chunks.add(chunk);
        }
    }

    /**
     * Returns true if this script is of the form <pubkey> OP_CHECKSIG. This form was originally
intended for transactions
```

```
     * where the peers talked to each other directly via TCP/IP, but has fallen out of favor with time due to that mode
     * of operation being susceptible to man-in-the-middle attacks. It is still used in coinbase outputs and can be
     * useful more exotic types of transaction, but today most payments are to addresses.
     * <p>
     * (scriptPublicKey)
     */
    public boolean isSentToRawPubKey() {
        return chunks.size() == 2 && chunks.get(1).equalsOpCode(OP_CHECKSIG) &&
                !chunks.get(0).isOpCode() && chunks.get(0).data.length > 1;
    }

    /**
     * Returns true if this script is of the form DUP HASH160 <pubkey hash> EQUALVERIFY CHECKSIG, ie, payment to an
     * address like 1VayNert3x1KzbpzMGt2qdqrAThiRovi8. This form was originally intended for the case where you wish
     * to send somebody money with a written code because their node is offline, but over time has become the standard
     * way to make payments due to the short and recognizable base58 form addresses come in.
     * <p>
     *
     */
    public boolean isSentToAddress() {
        return chunks.size() == 5 &&
                chunks.get(0).equalsOpCode(OP_DUP) &&
                chunks.get(1).equalsOpCode(OP_HASH160) &&
                chunks.get(2).data.length == Address.ADDRESS_LENGTH &&
                chunks.get(3).equalsOpCode(OP_EQUALVERIFY) &&
                chunks.get(4).equalsOpCode(OP_CHECKSIG);
    }

    /**
     * An alias for isPayToScriptHash.
     */
    @Deprecated
    public boolean isSentToP2SH() {
        return isPayToScriptHash();
    }

    /**
```

```
    * <p>If a program matches the standard template DUP HASH160 &lt;pubkey hash&gt;
EQUALVERIFY CHECKSIG
    * then this function retrieves the third element.
    * In this case, this is useful for fetching the destination address of a transaction.</p>
    *
    * <p>If a program matches the standard template HASH160 &lt;script hash&gt; EQUAL
    * then this function retrieves the second element.
    * In this case, this is useful for fetching the hash of the redeem script of a transaction.</p>
    *
    * <p>Otherwise it throws a ScriptException.</p>
    * <p>
    * HASH
    */
   public byte[] getPubKeyHash() throws ScriptException {
      if (isSentToAddress()) {
         return chunks.get(2).data;
      } else if (isPayToScriptHash()) {
         return chunks.get(1).data;
      } else {
         throw new ScriptException("Script not in the standard scriptPubKey form");
      }
   }

   /**
    * Returns the public key in this script. If a script contains two constants and nothing else, it is
assumed to
    * be a scriptSig (input) for a pay-to-address output and the second constant is returned (the first
is the
    * signature). If a script contains a constant and an OP_CHECKSIG opcode, the constant is
returned as it is
    * assumed to be a direct pay-to-key scriptPubKey (output) and the first constant is the public
key.
    *
    * @throws ScriptException if the script is none of the named forms.
    *                         <p>
    *
    */
   public byte[] getPubKey() throws ScriptException {
      if (chunks.size() != 2) {
         throw new ScriptException("Script not of right size, expecting 2 but got " + chunks.size());
      }
      final ScriptChunk chunk0 = chunks.get(0);
```

```
        final byte[] chunk0data = chunk0.data;
        final ScriptChunk chunk1 = chunks.get(1);
        final byte[] chunk1data = chunk1.data;
        if (chunk0data != null && chunk0data.length > 2 && chunk1data != null && chunk1data.length
> 2) {
            // If we have two large constants assume the input to a pay-to-address output.
            return chunk1data;
        } else if (chunk1.equalsOpCode(OP_CHECKSIG) && chunk0data != null &&
chunk0data.length > 2) {
            // A large constant followed by an OP_CHECKSIG is the key.
            return chunk0data;
        } else {
            throw new ScriptException("Script did not match expected form: " + this);
        }
    }

    /**
     * Retrieves the sender public key from a LOCKTIMEVERIFY transaction
     *
     * @return
     * @throws ScriptException
     */
    public byte[] getCLTVPaymentChannelSenderPubKey() throws ScriptException {
        if (!isSentToCLTVPaymentChannel()) {
            throw new ScriptException("Script not a standard CHECKLOCKTIMVERIFY transaction: "
+ this);
        }
        return chunks.get(8).data;
    }

    /**
     * Retrieves the recipient public key from a LOCKTIMEVERIFY transaction
     *
     * @return
     * @throws ScriptException
     */
    public byte[] getCLTVPaymentChannelRecipientPubKey() throws ScriptException {
        if (!isSentToCLTVPaymentChannel()) {
            throw new ScriptException("Script not a standard CHECKLOCKTIMVERIFY transaction: "
+ this);
        }
        return chunks.get(1).data;
```

```
    }

    public BigInteger getCLTVPaymentChannelExpiry() {
        if (!isSentToCLTVPaymentChannel()) {
            throw new ScriptException("Script not a standard CHECKLOCKTIMEVERIFY transaction:
" + this);
        }
        return castToBigInteger(chunks.get(4).data, 5);
    }

    /**
     * For 2-element [input] scripts assumes that the paid-to-address can be derived from the public
key.
     * The concept of a "from address" isn't well defined in Bitcoin and you should not assume the
sender of a
     * transaction can actually receive coins on it. This method may be removed in future.
     *
     *
     */
    /*@Deprecated
    public Address getFromAddress(NetworkParameters params) throws ScriptException {
        return new Address(params, Utils.sha256hash160(getPubKey()));
    }*/

    /**
     * Gets the destination address from this script, if it's in the required form (see getPubKey).
     *
     */
    /*public Address getToAddress(NetworkParameters params) throws ScriptException {
        return getToAddress(params, false);
    }*/

    /**
     * Gets the destination address from this script, if it's in the required form (see getPubKey).
     *
     * @param forcePayToPubKey
     *            If true, allow payToPubKey to be casted to the corresponding address. This is useful if
you prefer
     *            showing addresses rather than pubkeys.
     */
    /*public Address getToAddress(NetworkParameters params, boolean forcePayToPubKey)
throws ScriptException {
```

```
        if (isSentToAddress())
            return new Address(params, getPubKeyHash());
        else if (isPayToScriptHash())
            return Address.fromP2SHScript(params, this);
        else if (forcePayToPubKey && isSentToRawPubKey())
            return ECKey.fromPublicOnly(getPubKey()).toAddress(params);
        else
            throw new ScriptException("Cannot cast this script to a pay-to-address type");
    }*/

    ////////////////////// Interface for writing scripts from scratch ////////////////////////////////

    /**
     * Writes out the given byte buffer to the output stream with the correct opcode prefix
     * To write an integer call writeBytes(out,
SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(val, false)));
     * <p>
     *
     */
    public static void writeBytes(OutputStream os, byte[] buf) throws IOException {
        if (buf.length < OP_PUSHDATA1) {
            os.write(buf.length);
            os.write(buf);
        } else if (buf.length < 256) {
            os.write(OP_PUSHDATA1);
            os.write(buf.length);
            os.write(buf);
        } else if (buf.length < 65536) {
            os.write(OP_PUSHDATA2);
            os.write(0xFF & (buf.length));
            os.write(0xFF & (buf.length >> 8));
            os.write(buf);
        } else {
            throw new RuntimeException("Unimplemented");
        }
    }

    /**
     * Creates a program that requires at least N of the given keys to sign, using
OP_CHECKMULTISIG.
     * OutputScript/scriptPublicKry
     */
```

```
    public static byte[] createMultiSigOutputScript(int threshold, List<ECKey> pubkeys) {
        checkArgument(threshold > 0);
        checkArgument(threshold <= pubkeys.size());
        checkArgument(pubkeys.size() <= 16);  // That's the max we can represent with a single opcode.
        if (pubkeys.size() > 3) {
            log.warn("Creating a multi-signature output that is non-standard: {} pubkeys, should be <= 3", pubkeys.size());
        }
        try {
            ByteArrayOutputStream bits = new ByteArrayOutputStream();
            bits.write(encodeToOpN(threshold));
            for (ECKey key : pubkeys) {
                writeBytes(bits, key.getPubKey());
            }
            bits.write(encodeToOpN(pubkeys.size()));
            bits.write(OP_CHECKMULTISIG);
            return bits.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException(e);  // Cannot happen.
        }
    }

    /**
     * inputScript/scriptSig
     **/
    public static byte[] createInputScript(byte[] signature, byte[] pubkey) {
        try {
            // TODO: Do this by creating a Script *first* then having the script reassemble itself into bytes.
            ByteArrayOutputStream bits = new UnsafeByteArrayOutputStream(signature.length + pubkey.length + 2);
            writeBytes(bits, signature);
            writeBytes(bits, pubkey);
            return bits.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * inputScript/scriptSig
```

```
     **/
    public static byte[] createInputScript(byte[] signature) {
        try {
            // TODO: Do this by creating a Script *first* then having the script reassemble itself into bytes.
            ByteArrayOutputStream bits = new UnsafeByteArrayOutputStream(signature.length + 2);
            writeBytes(bits, signature);
            return bits.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Creates an incomplete scriptSig that, once filled with signatures, can redeem output containing this scriptPubKey.
     * Instead of the signatures resulting script has OP_0.
     * Having incomplete input script allows to pass around partially signed tx.
     * It is expected that this program later on will be updated with proper signatures.
     * <p>
     * Script
     */
    public Script createEmptyInputScript(@Nullable ECKey key, @Nullable Script redeemScript) {
        if (isSentToAddress()) {
            checkArgument(key != null, "Key required to create pay-to-address input script");
            return ScriptBuilder.createInputScript(null, key);
        } else if (isSentToRawPubKey()) {
            return ScriptBuilder.createInputScript(null);
        } else if (isPayToScriptHash()) {
            checkArgument(redeemScript != null, "Redeem script required to create P2SH input script");
            return ScriptBuilder.createP2SHMultiSigInputScript(null, redeemScript);
        } else {
            throw new ScriptException("Do not understand script type: " + this);
        }
    }

    /**
     * Returns a copy of the given scriptSig with the signature inserted in the given position.
     */
    public Script getScriptSigWithSignature(Script scriptSig, byte[] sigBytes, int index) {
        int sigsPrefixCount = 0;
```

```
        int sigsSuffixCount = 0;
        if (isPayToScriptHash()) {
            sigsPrefixCount = 1; // OP_0 <sig>* <redeemScript>
            sigsSuffixCount = 1;
        } else if (isSentToMultiSig()) {
            sigsPrefixCount = 1; // OP_0 <sig>*
        } else if (isSentToAddress()) {
            sigsSuffixCount = 1; // <sig> <pubkey>
        }
        return ScriptBuilder.updateScriptWithSignature(scriptSig, sigBytes, index, sigsPrefixCount,
sigsSuffixCount);
    }


    /**
     * Returns the index where a signature by the key should be inserted.  Only applicable to
     * a P2SH scriptSig.
     */
    public int getSigInsertionIndex(Sha256Hash hash, ECKey signingKey) {
        // Iterate over existing signatures, skipping the initial OP_0, the final redeem script
        // and any placeholder OP_0 sigs.
        List<ScriptChunk> existingChunks = chunks.subList(1, chunks.size() - 1);
        ScriptChunk redeemScriptChunk = chunks.get(chunks.size() - 1);
        checkNotNull(redeemScriptChunk.data);
        Script redeemScript = new Script(redeemScriptChunk.data);

        int sigCount = 0;
        int myIndex = redeemScript.findKeyInRedeem(signingKey);
        for (ScriptChunk chunk : existingChunks) {
            if (chunk.opcode == OP_0) {
                // OP_0, skip
            } else {
                checkNotNull(chunk.data);
                if (myIndex < redeemScript.findSigInRedeem(chunk.data, hash)) {
                    return sigCount;
                }
                sigCount++;
            }
        }
        return sigCount;
    }
```

```
    private int findKeyInRedeem(ECKey key) {
        checkArgument(chunks.get(0).isOpCode()); // P2SH scriptSig
        int numKeys = Script.decodeFromOpN(chunks.get(chunks.size() - 2).opcode);
        for (int i = 0; i < numKeys; i++) {
            if (Arrays.equals(chunks.get(1 + i).data, key.getPubKey())) {
                return i;
            }
        }

        throw new IllegalStateException("Could not find matching key " + key.toString() + " in script "
+ this);
    }

    /**
     * Returns a list of the keys required by this script, assuming a multi-sig script.
     *
     * @throws ScriptException if the script type is not understood or is pay to address or is P2SH
(run this method on the "Redeem script" instead).
     */
    public List<ECKey> getPubKeys() {
        if (!isSentToMultiSig()) {
            throw new ScriptException("Only usable for multisig scripts.");
        }

        ArrayList<ECKey> result = Lists.newArrayList();
        int numKeys = Script.decodeFromOpN(chunks.get(chunks.size() - 2).opcode);
        for (int i = 0; i < numKeys; i++) {
            result.add(ECKey.fromPublicOnly(chunks.get(1 + i).data));
        }
        return result;
    }

    private int findSigInRedeem(byte[] signatureBytes, Sha256Hash hash) {
        checkArgument(chunks.get(0).isOpCode()); // P2SH scriptSig
        int numKeys = Script.decodeFromOpN(chunks.get(chunks.size() - 2).opcode);
       /* TransactionSignature signature = TransactionSignature.decodeFromBitcoin(signatureBytes,
true);
        for (int i = 0 ; i < numKeys ; i++) {
            if (ECKey.fromPublicOnly(chunks.get(i + 1).data).verify(hash, signature)) {
                return i;
            }
        }*/
```

```
        throw new IllegalStateException("Could not find matching key for signature on " +
hash.toString() + " sig " + Hex.encode(signatureBytes));
    }


    ////////////////////// Interface used during verification of transactions/blocks ////////////////////////////////

    private static int getSigOpCount(List<ScriptChunk> chunks, boolean accurate) throws
ScriptException {
        int sigOps = 0;
        int lastOpCode = OP_INVALIDOPCODE;
        for (ScriptChunk chunk : chunks) {
            if (chunk.isOpCode()) {
                switch (chunk.opcode) {
                    case OP_CHECKSIG:
                    case OP_CHECKSIGVERIFY:
                        sigOps++;
                        break;
                    case OP_CHECKMULTISIG:
                    case OP_CHECKMULTISIGVERIFY:
                        if (accurate && lastOpCode >= OP_1 && lastOpCode <= OP_16) {
                            sigOps += decodeFromOpN(lastOpCode);
                        } else {
                            sigOps += 20;
                        }
                        break;
                    default:
                        break;
                }
                lastOpCode = chunk.opcode;
            }
        }
        return sigOps;
    }

    static int decodeFromOpN(int opcode) {
        checkArgument((opcode == OP_0 || opcode == OP_1NEGATE) || (opcode >= OP_1 &&
opcode <= OP_16), "decodeFromOpN called on non OP_N opcode");
        if (opcode == OP_0) {
            return 0;
        } else if (opcode == OP_1NEGATE) {
```

```
            return -1;
        } else {
            return opcode + 1 - OP_1;
        }
    }

    static int encodeToOpN(int value) {
        checkArgument(value >= -1 && value <= 16, "encodeToOpN called for " + value + " which we cannot encode in an opcode.");
        if (value == 0) {
            return OP_0;
        } else if (value == -1) {
            return OP_1NEGATE;
        } else {
            return value - 1 + OP_1;
        }
    }

    /**
     * Gets the count of regular SigOps in the script program (counting multisig ops as 20)
     */
    public static int getSigOpCount(byte[] program) throws ScriptException {
        Script script = new Script();
        try {
            script.parse(program);
        } catch (ScriptException e) {
            // Ignore errors and count up to the parse-able length
        }
        return getSigOpCount(script.chunks, false);
    }

    /**
     * Gets the count of P2SH Sig Ops in the Script scriptSig
     */
    public static long getP2SHSigOpCount(byte[] scriptSig) throws ScriptException {
        Script script = new Script();
        try {
            script.parse(scriptSig);
        } catch (ScriptException e) {
            // Ignore errors and count up to the parse-able length
        }
        for (int i = script.chunks.size() - 1; i >= 0; i--) {
```

```
            if (!script.chunks.get(i).isOpCode()) {
                Script subScript = new Script();
                subScript.parse(script.chunks.get(i).data);
                return getSigOpCount(subScript.chunks, true);
            }
        }
        return 0;
    }

    /**
     * Returns number of signatures required to satisfy this script.
     */
    public int getNumberOfSignaturesRequiredToSpend() {
        if (isSentToMultiSig()) {
            // for N of M CHECKMULTISIG script we will need N signatures to spend
            ScriptChunk nChunk = chunks.get(0);
            return Script.decodeFromOpN(nChunk.opcode);
        } else if (isSentToAddress() || isSentToRawPubKey()) {
            // pay-to-address and pay-to-pubkey require single sig
            return 1;
        } else if (isPayToScriptHash()) {
            throw new IllegalStateException("For P2SH number of signatures depends on redeem
script");
        } else {
            throw new IllegalStateException("Unsupported script type");
        }
    }

    /**
     * Returns number of bytes required to spend this script. It accepts optional ECKey and
redeemScript that may
     * be required for certain types of script to estimate target size.
     */
    public int getNumberOfBytesRequiredToSpend(@Nullable ECKey pubKey, @Nullable Script
redeemScript) {
        if (isPayToScriptHash()) {
            // scriptSig: <sig> [sig] [sig...] <redeemscript>
            checkArgument(redeemScript != null, "P2SH script requires redeemScript to be spent");
            return redeemScript.getNumberOfSignaturesRequiredToSpend() * SIG_SIZE +
redeemScript.getProgram().length;
        } else if (isSentToMultiSig()) {
            // scriptSig: OP_0 <sig> [sig] [sig...]
```

```
            return getNumberOfSignaturesRequiredToSpend() * SIG_SIZE + 1;
        } else if (isSentToRawPubKey()) {
            // scriptSig: <sig>
            return SIG_SIZE;
        } else if (isSentToAddress()) {
            // scriptSig: <sig> <pubkey>
            int uncompressedPubKeySize = 65;
            return SIG_SIZE + (pubKey != null ? pubKey.getPubKey().length :
uncompressedPubKeySize);
        } else {
            throw new IllegalStateException("Unsupported script type");
        }
    }

    /**
     * <p>Whether or not this is a scriptPubKey representing a pay-to-script-hash output. In such
outputs, the logic that
     * controls reclamation is not actually in the output at all. Instead there's just a hash, and it's up
to the
     * spending input to provide a program matching that hash. This rule is "soft enforced" by the
network as it does
     * not exist in Bitcoin Core. It means blocks containing P2SH transactions that don't match
     * correctly are considered valid, but won't be mined upon, so they'll be rapidly re-orgd out of the
chain. This
     * logic is defined by <a href="https://github.com/bitcoin/bips/blob/master/bip-
0016.mediawiki">BIP 16</a>.</p>
     *
     * <p>bitcoinj does not support creation of P2SH transactions today. The goal of P2SH is to
allow short addresses
     * even for complex scripts (eg, multi-sig outputs) so they are convenient to work with in things
like QRcodes or
     * with copy/paste, and also to minimize the size of the unspent output set (which improves
performance of the
     * Bitcoin system).</p>
     */
    public boolean isPayToScriptHash() {
        // We have to check against the serialized form because BIP16 defines a P2SH output using
an exact byte
        // template, not the logical program structure. Thus you can have two programs that look
identical when
        // printed out but one is a P2SH script and the other isn't! :(
        byte[] program = getProgram();
```

```
        return program.length == 26 &&
                (program[0] & 0xff) == OP_HASH160 &&
                (program[1] & 0xff) == 0x17 &&
                (program[25] & 0xff) == OP_EQUAL;
    }

    /**
     * Returns whether this script matches the format used for multisig outputs: [n] [keys...] [m]
CHECKMULTISIG
     */
    public boolean isSentToMultiSig() {
        if (chunks.size() < 4) {
            return false;
        }
        ScriptChunk chunk = chunks.get(chunks.size() - 1);
        // Must end in OP_CHECKMULTISIG[VERIFY].
        if (!chunk.isOpCode()) {
            return false;
        }
        if (!(chunk.equalsOpCode(OP_CHECKMULTISIG) ||
chunk.equalsOpCode(OP_CHECKMULTISIGVERIFY))) {
            return false;
        }
        try {
            // Second to last chunk must be an OP_N opcode and there should be that many data
chunks (keys).
            ScriptChunk m = chunks.get(chunks.size() - 2);
            if (!m.isOpCode()) {
                return false;
            }
            int numKeys = decodeFromOpN(m.opcode);
            if (numKeys < 1 || chunks.size() != 3 + numKeys) {
                return false;
            }
            for (int i = 1; i < chunks.size() - 2; i++) {
                if (chunks.get(i).isOpCode()) {
                    return false;
                }
            }
            // First chunk must be an OP_N opcode too.
            if (decodeFromOpN(chunks.get(0).opcode) < 1) {
                return false;
```

```
        }
    } catch (IllegalArgumentException e) { // thrown by decodeFromOpN()
        return false;   // Not an OP_N opcode.
    }
    return true;
}

public boolean isSentToCLTVPaymentChannel() {
    if (chunks.size() != 10) {
        return false;
    }
    // Check that opcodes match the pre-determined format.
    if (!chunks.get(0).equalsOpCode(OP_IF)) {
        return false;
    }
    // chunk[1] = recipient pubkey
    if (!chunks.get(2).equalsOpCode(OP_CHECKSIGVERIFY)) {
        return false;
    }
    if (!chunks.get(3).equalsOpCode(OP_ELSE)) {
        return false;
    }
    // chunk[4] = locktime
    if (!chunks.get(5).equalsOpCode(OP_CHECKLOCKTIMEVERIFY)) {
        return false;
    }
    if (!chunks.get(6).equalsOpCode(OP_DROP)) {
        return false;
    }
    if (!chunks.get(7).equalsOpCode(OP_ENDIF)) {
        return false;
    }
    // chunk[8] = sender pubkey
    if (!chunks.get(9).equalsOpCode(OP_CHECKSIG)) {
        return false;
    }
    return true;
}

private static boolean equalsRange(byte[] a, int start, byte[] b) {
    if (start + b.length > a.length) {
        return false;
```

```
        }
        for (int i = 0; i < b.length; i++) {
            if (a[i + start] != b[i]) {
                return false;
            }
        }
        return true;
    }

    /**
     * Returns the script bytes of inputScript with all instances of the specified script object removed
     */
    public static byte[] removeAllInstancesOf(byte[] inputScript, byte[] chunkToRemove) {
        // We usually don't end up removing anything
        UnsafeByteArrayOutputStream bos = new
UnsafeByteArrayOutputStream(inputScript.length);

        int cursor = 0;
        while (cursor < inputScript.length) {
            boolean skip = equalsRange(inputScript, cursor, chunkToRemove);

            int opcode = inputScript[cursor++] & 0xFF;
            int additionalBytes = 0;
            if (opcode >= 0 && opcode < OP_PUSHDATA1) {
                additionalBytes = opcode;
            } else if (opcode == OP_PUSHDATA1) {
                additionalBytes = (0xFF & inputScript[cursor]) + 1;
            } else if (opcode == OP_PUSHDATA2) {
                additionalBytes = ((0xFF & inputScript[cursor]) |
                        ((0xFF & inputScript[cursor + 1]) << 8)) + 2;
            } else if (opcode == OP_PUSHDATA4) {
                additionalBytes = ((0xFF & inputScript[cursor]) |
                        ((0xFF & inputScript[cursor + 1]) << 8) |
                        ((0xFF & inputScript[cursor + 1]) << 16) |
                        ((0xFF & inputScript[cursor + 1]) << 24)) + 4;
            }
            if (!skip) {
                try {
                    bos.write(opcode);
                    bos.write(Arrays.copyOfRange(inputScript, cursor, cursor + additionalBytes));
                } catch (IOException e) {
                    throw new RuntimeException(e);
```

```
            }
        }
        cursor += additionalBytes;
    }
    return bos.toByteArray();
}

/**
 * Returns the script bytes of inputScript with all instances of the given op code removed
 */
public static byte[] removeAllInstancesOfOp(byte[] inputScript, int opCode) {
    return removeAllInstancesOf(inputScript, new byte[]{(byte) opCode});
}

////////////////////// Script verification and helpers ////////////////////////////////

private static boolean castToBool(byte[] data) {
    for (int i = 0; i < data.length; i++) {
        // "Can be negative zero" - Bitcoin Core (see OpenSSL's BN_bn2mpi)
        if (data[i] != 0) {
            return !(i == data.length - 1 && (data[i] & 0xFF) == 0x80);
        }
    }
    return false;
}

/**
 * Cast a script chunk to a BigInteger.
 * <p>
 * sizes.
 *
 * @throws ScriptException if the chunk is longer than 4 bytes.
 */
private static BigInteger castToBigInteger(byte[] chunk) throws ScriptException {
    if (chunk.length > 4) {
        throw new ScriptException("Script attempted to use an integer larger than 4 bytes");
    }
    return SerializeUtils.decodeMPI(SerializeUtils.reverseBytes(chunk), false);
}

/**
 * Cast a script chunk to a BigInteger. Normally you would want
```

```
     * the normal maximum length does not apply (i.e. CHECKLOCKTIMEVERIFY).
     *
     * @param maxLength the maximum length in bytes.
     * @throws ScriptException if the chunk is longer than the specified maximum.
     */
    private static BigInteger castToBigInteger(final byte[] chunk, final int maxLength) throws
ScriptException {
        if (chunk.length > maxLength) {
            throw new ScriptException("Script attempted to use an integer larger than "
                    + maxLength + " bytes");
        }
        return SerializeUtils.decodeMPI(SerializeUtils.reverseBytes(chunk), false);
    }

    public boolean isOpReturn() {
        return chunks.size() > 0 && chunks.get(0).equalsOpCode(OP_RETURN);
    }

    /**
     * Exposes the script interpreter. Normally you should not use this directly, instead use
     * instead.
     */
    @Deprecated
    public static void executeScript(@Nullable Transaction txContainingThis, long index,
                                     Script script, LinkedList<byte[]> stack, boolean enforceNullDummy) throws
ScriptException {
        final EnumSet<VerifyFlag> flags = enforceNullDummy
                ? EnumSet.of(VerifyFlag.NULLDUMMY)
                : EnumSet.noneOf(VerifyFlag.class);

        executeScript(txContainingThis, index, script, stack, flags);
    }

    /**
     * Exposes the script interpreter. Normally you should not use this directly, instead use
     * is useful if you need more precise control or access to the final state of the stack. This
interface is very
     * likely to change in future.
     *
     * @param txContainingThis
     * @param index
     * @param script
```

```
     * @param stack
     * @param verifyFlags
     **/
    public static void executeScript(@Nullable Transaction txContainingThis, long index,
                                     Script script, LinkedList<byte[]> stack, Set<VerifyFlag> verifyFlags) throws
ScriptException {
        int opCount = 0;
        int lastCodeSepLocation = 0;

        LinkedList<byte[]> altstack = new LinkedList<byte[]>();
        LinkedList<Boolean> ifStack = new LinkedList<Boolean>();

        for (ScriptChunk chunk : script.chunks) {
            boolean shouldExecute = !ifStack.contains(false);

            if (chunk.opcode == OP_0) {
                if (!shouldExecute) {
                    continue;
                }

                stack.add(new byte[]{});
            } else if (!chunk.isOpCode()) {
                if (chunk.data.length > MAX_SCRIPT_ELEMENT_SIZE) {
                    throw new ScriptException("Attempted to push a data string larger than 520 bytes");
                }

                if (!shouldExecute) {
                    continue;
                }

                stack.add(chunk.data);
            } else {
                int opcode = chunk.opcode;
                if (opcode > OP_16) {
                    opCount++;
                    if (opCount > 201) {
                        throw new ScriptException("More script operations than is allowed");
                    }
                }

                if (opcode == OP_VERIF || opcode == OP_VERNOTIF) {
                    throw new ScriptException("Script included OP_VERIF or OP_VERNOTIF");
```

```
        }

        if (opcode == OP_CAT || opcode == OP_SUBSTR || opcode == OP_LEFT || opcode ==
OP_RIGHT ||
                opcode == OP_INVERT || opcode == OP_AND || opcode == OP_OR || opcode ==
OP_XOR ||
                opcode == OP_2MUL || opcode == OP_2DIV || opcode == OP_MUL || opcode ==
OP_DIV ||
                opcode == OP_MOD || opcode == OP_LSHIFT || opcode == OP_RSHIFT) {
            throw new ScriptException("Script included a disabled Script Op.");
        }

        switch (opcode) {
            case OP_IF:
                if (!shouldExecute) {
                    ifStack.add(false);
                    continue;
                }
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_IF on an empty stack");
                }
                ifStack.add(castToBool(stack.pollLast()));
                continue;
            case OP_NOTIF:
                if (!shouldExecute) {
                    ifStack.add(false);
                    continue;
                }
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_NOTIF on an empty stack");
                }
                ifStack.add(!castToBool(stack.pollLast()));
                continue;
            case OP_ELSE:
                if (ifStack.isEmpty()) {
                    throw new ScriptException("Attempted OP_ELSE without OP_IF/NOTIF");
                }
                ifStack.add(!ifStack.pollLast());
                continue;
            case OP_ENDIF:
                if (ifStack.isEmpty()) {
                    throw new ScriptException("Attempted OP_ENDIF without OP_IF/NOTIF");
```

```
                }
                ifStack.pollLast();
                continue;
            }

            if (!shouldExecute) {
                continue;
            }

            switch (opcode) {
                // OP_0 is no opcode
                case OP_1NEGATE:
stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.ONE.negate(), false)));
                  break;
                case OP_1:
                case OP_2:
                case OP_3:
                case OP_4:
                case OP_5:
                case OP_6:
                case OP_7:
                case OP_8:
                case OP_9:
                case OP_10:
                case OP_11:
                case OP_12:
                case OP_13:
                case OP_14:
                case OP_15:
                case OP_16:
stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.valueOf(decodeFromO
pN(opcode)), false)));
                    break;
                case OP_NOP:
                    break;
                case OP_VERIFY:
                    if (stack.size() < 1) {
                        throw new ScriptException("Attempted OP_VERIFY on an empty stack");
                    }
                    if (!castToBool(stack.pollLast())) {
                        throw new ScriptException("OP_VERIFY failed");
                    }
```

```
                break;
            case OP_RETURN:
                throw new ScriptException("Script called OP_RETURN");
            case OP_TOALTSTACK:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_TOALTSTACK on an empty stack");
                }
                altstack.add(stack.pollLast());
                break;
            case OP_FROMALTSTACK:
                if (altstack.size() < 1) {
                    throw new ScriptException("Attempted OP_FROMALTSTACK on an empty
altstack");
                }
                stack.add(altstack.pollLast());
                break;
            case OP_2DROP:
                if (stack.size() < 2) {
                    throw new ScriptException("Attempted OP_2DROP on a stack with size < 2");
                }
                stack.pollLast();
                stack.pollLast();
                break;
            case OP_2DUP:
                if (stack.size() < 2) {
                    throw new ScriptException("Attempted OP_2DUP on a stack with size < 2");
                }
                Iterator<byte[]> it2DUP = stack.descendingIterator();
                byte[] OP2DUPtmpChunk2 = it2DUP.next();
                stack.add(it2DUP.next());
                stack.add(OP2DUPtmpChunk2);
                break;
            case OP_3DUP:
                if (stack.size() < 3) {
                    throw new ScriptException("Attempted OP_3DUP on a stack with size < 3");
                }
                Iterator<byte[]> it3DUP = stack.descendingIterator();
                byte[] OP3DUPtmpChunk3 = it3DUP.next();
                byte[] OP3DUPtmpChunk2 = it3DUP.next();
                stack.add(it3DUP.next());
                stack.add(OP3DUPtmpChunk2);
                stack.add(OP3DUPtmpChunk3);
```

```
            break;
        case OP_2OVER:
            if (stack.size() < 4) {
                throw new ScriptException("Attempted OP_2OVER on a stack with size < 4");
            }
            Iterator<byte[]> it2OVER = stack.descendingIterator();
            it2OVER.next();
            it2OVER.next();
            byte[] OP2OVERtmpChunk2 = it2OVER.next();
            stack.add(it2OVER.next());
            stack.add(OP2OVERtmpChunk2);
            break;
        case OP_2ROT:
            if (stack.size() < 6) {
                throw new ScriptException("Attempted OP_2ROT on a stack with size < 6");
            }
            byte[] OP2ROTtmpChunk6 = stack.pollLast();
            byte[] OP2ROTtmpChunk5 = stack.pollLast();
            byte[] OP2ROTtmpChunk4 = stack.pollLast();
            byte[] OP2ROTtmpChunk3 = stack.pollLast();
            byte[] OP2ROTtmpChunk2 = stack.pollLast();
            byte[] OP2ROTtmpChunk1 = stack.pollLast();
            stack.add(OP2ROTtmpChunk3);
            stack.add(OP2ROTtmpChunk4);
            stack.add(OP2ROTtmpChunk5);
            stack.add(OP2ROTtmpChunk6);
            stack.add(OP2ROTtmpChunk1);
            stack.add(OP2ROTtmpChunk2);
            break;
        case OP_2SWAP:
            if (stack.size() < 4) {
                throw new ScriptException("Attempted OP_2SWAP on a stack with size < 4");
            }
            byte[] OP2SWAPtmpChunk4 = stack.pollLast();
            byte[] OP2SWAPtmpChunk3 = stack.pollLast();
            byte[] OP2SWAPtmpChunk2 = stack.pollLast();
            byte[] OP2SWAPtmpChunk1 = stack.pollLast();
            stack.add(OP2SWAPtmpChunk3);
            stack.add(OP2SWAPtmpChunk4);
            stack.add(OP2SWAPtmpChunk1);
            stack.add(OP2SWAPtmpChunk2);
            break;
```

```
                case OP_IFDUP:
                    if (stack.size() < 1) {
                        throw new ScriptException("Attempted OP_IFDUP on an empty stack");
                    }
                    if (castToBool(stack.getLast())) {
                        stack.add(stack.getLast());
                    }
                    break;
                case OP_DEPTH:
stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.valueOf(stack.size()), f
alse)));
                    break;
                case OP_DROP:
                    if (stack.size() < 1) {
                        throw new ScriptException("Attempted OP_DROP on an empty stack");
                    }
                    stack.pollLast();
                    break;
                case OP_DUP:
                    if (stack.size() < 1) {
                        throw new ScriptException("Attempted OP_DUP on an empty stack");
                    }
                    stack.add(stack.getLast());
                    break;
                case OP_NIP:
                    if (stack.size() < 2) {
                        throw new ScriptException("Attempted OP_NIP on a stack with size < 2");
                    }
                    byte[] OPNIPtmpChunk = stack.pollLast();
                    stack.pollLast();
                    stack.add(OPNIPtmpChunk);
                    break;
                case OP_OVER:
                    if (stack.size() < 2) {
                        throw new ScriptException("Attempted OP_OVER on a stack with size < 2");
                    }
                    Iterator<byte[]> itOVER = stack.descendingIterator();
                    itOVER.next();
                    stack.add(itOVER.next());
                    break;
                case OP_PICK:
                case OP_ROLL:
```

```
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_PICK/OP_ROLL on an empty
stack");
                }
                long val = castToBigInteger(stack.pollLast()).longValue();
                if (val < 0 || val >= stack.size()) {
                    throw new ScriptException("OP_PICK/OP_ROLL attempted to get data deeper
than stack size");
                }
                Iterator<byte[]> itPICK = stack.descendingIterator();
                for (long i = 0; i < val; i++) {
                    itPICK.next();
                }
                byte[] OPROLLtmpChunk = itPICK.next();
                if (opcode == OP_ROLL) {
                    itPICK.remove();
                }
                stack.add(OPROLLtmpChunk);
                break;
            case OP_ROT:
                if (stack.size() < 3) {
                    throw new ScriptException("Attempted OP_ROT on a stack with size < 3");
                }
                byte[] OPROTtmpChunk3 = stack.pollLast();
                byte[] OPROTtmpChunk2 = stack.pollLast();
                byte[] OPROTtmpChunk1 = stack.pollLast();
                stack.add(OPROTtmpChunk2);
                stack.add(OPROTtmpChunk3);
                stack.add(OPROTtmpChunk1);
                break;
            case OP_SWAP:
            case OP_TUCK:
                if (stack.size() < 2) {
                    throw new ScriptException("Attempted OP_SWAP on a stack with size < 2");
                }
                byte[] OPSWAPtmpChunk2 = stack.pollLast();
                byte[] OPSWAPtmpChunk1 = stack.pollLast();
                stack.add(OPSWAPtmpChunk2);
                stack.add(OPSWAPtmpChunk1);
                if (opcode == OP_TUCK) {
                    stack.add(OPSWAPtmpChunk2);
                }
```

```
                break;
            case OP_CAT:
            case OP_SUBSTR:
            case OP_LEFT:
            case OP_RIGHT:
                throw new ScriptException("Attempted to use disabled Script Op.");
            case OP_SIZE:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_SIZE on an empty stack");
                }
stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.valueOf(stack.getLast()
.length), false)));
                break;
            case OP_INVERT:
            case OP_AND:
            case OP_OR:
            case OP_XOR:
                throw new ScriptException("Attempted to use disabled Script Op.");
            case OP_EQUAL:
                if (stack.size() < 2) {
                    throw new ScriptException("Attempted OP_EQUAL on a stack with size < 2");
                }
                stack.add(AddressTool.checkPublicKeyHash(stack.pollLast(), stack.pollLast()) ?
new byte[]{1} : new byte[]{});
                break;
            case OP_EQUALVERIFY:
                if (stack.size() < 2) {
                    throw new ScriptException("Attempted OP_EQUALVERIFY on a stack with size
< 2");
                }
            /*if (!Arrays.equals(stack.pollLast(), stack.pollLast()))
                throw new ScriptException("OP_EQUALVERIFY: non-equal data");*/
                if (!AddressTool.checkPublicKeyHash(stack.pollLast(), stack.pollLast())) {
                    throw new ScriptException("OP_EQUALVERIFY: non-equal data");
                }
                break;
            case OP_1ADD:
            case OP_1SUB:
            case OP_NEGATE:
            case OP_ABS:
            case OP_NOT:
            case OP_0NOTEQUAL:
```

```
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted a numeric op on an empty stack");
                }
                BigInteger numericOPnum = castToBigInteger(stack.pollLast());

                switch (opcode) {
                    case OP_1ADD:
                        numericOPnum = numericOPnum.add(BigInteger.ONE);
                        break;
                    case OP_1SUB:
                        numericOPnum = numericOPnum.subtract(BigInteger.ONE);
                        break;
                    case OP_NEGATE:
                        numericOPnum = numericOPnum.negate();
                        break;
                    case OP_ABS:
                        if (numericOPnum.signum() < 0) {
                            numericOPnum = numericOPnum.negate();
                        }
                        break;
                    case OP_NOT:
                        if (numericOPnum.equals(BigInteger.ZERO)) {
                            numericOPnum = BigInteger.ONE;
                        } else {
                            numericOPnum = BigInteger.ZERO;
                        }
                        break;
                    case OP_0NOTEQUAL:
                        if (numericOPnum.equals(BigInteger.ZERO)) {
                            numericOPnum = BigInteger.ZERO;
                        } else {
                            numericOPnum = BigInteger.ONE;
                        }
                        break;
                    default:
                        throw new AssertionError("Unreachable");
                }

                stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(numericOPnum,
false)));
                break;
            case OP_2MUL:
```

```
                case OP_2DIV:
                    throw new ScriptException("Attempted to use disabled Script Op.");
                case OP_ADD:
                case OP_SUB:
                case OP_BOOLAND:
                case OP_BOOLOR:
                case OP_NUMEQUAL:
                case OP_NUMNOTEQUAL:
                case OP_LESSTHAN:
                case OP_GREATERTHAN:
                case OP_LESSTHANOREQUAL:
                case OP_GREATERTHANOREQUAL:
                case OP_MIN:
                case OP_MAX:
                    if (stack.size() < 2) {
                        throw new ScriptException("Attempted a numeric op on a stack with size < 2");
                    }
                    BigInteger numericOPnum2 = castToBigInteger(stack.pollLast());
                    BigInteger numericOPnum1 = castToBigInteger(stack.pollLast());

                    BigInteger numericOPresult;
                    switch (opcode) {
                        case OP_ADD:
                            numericOPresult = numericOPnum1.add(numericOPnum2);
                            break;
                        case OP_SUB:
                            numericOPresult = numericOPnum1.subtract(numericOPnum2);
                            break;
                        case OP_BOOLAND:
                            if (!numericOPnum1.equals(BigInteger.ZERO) &&
!numericOPnum2.equals(BigInteger.ZERO)) {
                                numericOPresult = BigInteger.ONE;
                            } else {
                                numericOPresult = BigInteger.ZERO;
                            }
                            break;
                        case OP_BOOLOR:
                            if (!numericOPnum1.equals(BigInteger.ZERO) ||
!numericOPnum2.equals(BigInteger.ZERO)) {
                                numericOPresult = BigInteger.ONE;
                            } else {
                                numericOPresult = BigInteger.ZERO;
```

```
        }
        break;
    case OP_NUMEQUAL:
        if (numericOPnum1.equals(numericOPnum2)) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
        }
        break;
    case OP_NUMNOTEQUAL:
        if (!numericOPnum1.equals(numericOPnum2)) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
        }
        break;
    case OP_LESSTHAN:
        if (numericOPnum1.compareTo(numericOPnum2) < 0) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
        }
        break;
    case OP_GREATERTHAN:
        if (numericOPnum1.compareTo(numericOPnum2) > 0) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
        }
        break;
    case OP_LESSTHANOREQUAL:
        if (numericOPnum1.compareTo(numericOPnum2) <= 0) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
        }
        break;
    case OP_GREATERTHANOREQUAL:
        if (numericOPnum1.compareTo(numericOPnum2) >= 0) {
            numericOPresult = BigInteger.ONE;
        } else {
            numericOPresult = BigInteger.ZERO;
```

```
                    }
                    break;
                case OP_MIN:
                    if (numericOPnum1.compareTo(numericOPnum2) < 0) {
                        numericOPresult = numericOPnum1;
                    } else {
                        numericOPresult = numericOPnum2;
                    }
                    break;
                case OP_MAX:
                    if (numericOPnum1.compareTo(numericOPnum2) > 0) {
                        numericOPresult = numericOPnum1;
                    } else {
                        numericOPresult = numericOPnum2;
                    }
                    break;
                default:
                    throw new RuntimeException("Opcode switched at runtime?");
            }

            stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(numericOPresult,
false)));
            break;
        case OP_MUL:
        case OP_DIV:
        case OP_MOD:
        case OP_LSHIFT:
        case OP_RSHIFT:
            throw new ScriptException("Attempted to use disabled Script Op.");
        case OP_NUMEQUALVERIFY:
            if (stack.size() < 2) {
                throw new ScriptException("Attempted OP_NUMEQUALVERIFY on a stack with
size < 2");
            }
            BigInteger OPNUMEQUALVERIFYnum2 = castToBigInteger(stack.pollLast());
            BigInteger OPNUMEQUALVERIFYnum1 = castToBigInteger(stack.pollLast());

            if (!OPNUMEQUALVERIFYnum1.equals(OPNUMEQUALVERIFYnum2)) {
                throw new ScriptException("OP_NUMEQUALVERIFY failed");
            }
            break;
        case OP_WITHIN:
```

```
                if (stack.size() < 3) {
                    throw new ScriptException("Attempted OP_WITHIN on a stack with size < 3");
                }
                BigInteger OPWITHINnum3 = castToBigInteger(stack.pollLast());
                BigInteger OPWITHINnum2 = castToBigInteger(stack.pollLast());
                BigInteger OPWITHINnum1 = castToBigInteger(stack.pollLast());
                if (OPWITHINnum2.compareTo(OPWITHINnum1) <= 0 &&
OPWITHINnum1.compareTo(OPWITHINnum3) < 0) {
                    stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.ONE,
false)));
                } else {
stack.add(SerializeUtils.reverseBytes(SerializeUtils.encodeMPI(BigInteger.ZERO, false)));
        }
                break;
            case OP_RIPEMD160:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_RIPEMD160 on an empty stack");
                }
                RIPEMD160Digest digest = new RIPEMD160Digest();
                byte[] dataToHash = stack.pollLast();
                digest.update(dataToHash, 0, dataToHash.length);
                byte[] ripmemdHash = new byte[20];
                digest.doFinal(ripmemdHash, 0);
                stack.add(ripmemdHash);
                break;
            case OP_SHA1:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_SHA1 on an empty stack");
                }
                try {
                    stack.add(MessageDigest.getInstance("SHA-1").digest(stack.pollLast()));
                } catch (NoSuchAlgorithmException e) {
                    throw new RuntimeException(e);  // Cannot happen.
                }
                break;
            case OP_SHA256:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_SHA256 on an empty stack");
                }
                stack.add(Sha256Hash.hash(stack.pollLast()));
                break;
            case OP_HASH160:
```

```
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_HASH160 on an empty stack");
                }
                stack.add(SerializeUtils.sha256hash160(stack.pollLast()));
                break;
            case OP_HASH256:
                if (stack.size() < 1) {
                    throw new ScriptException("Attempted OP_SHA256 on an empty stack");
                }
                stack.add(Sha256Hash.hashTwice(stack.pollLast()));
                break;
            case OP_CODESEPARATOR:
                lastCodeSepLocation = chunk.getStartLocationInProgram() + 1;
                break;
            case OP_CHECKSIG:
            case OP_CHECKSIGVERIFY:
                if (txContainingThis == null) {
                    throw new IllegalStateException("Script attempted signature check but no tx was
provided");
                }
                executeCheckSig(txContainingThis, (int) index, script, stack, lastCodeSepLocation,
opcode, verifyFlags);
                break;
            case OP_CHECKMULTISIG:
            case OP_CHECKMULTISIGVERIFY:
                if (txContainingThis == null) {
                    throw new IllegalStateException("Script attempted signature check but no tx was
provided");
                }
                opCount = executeMultiSig(txContainingThis, (int) index, script, stack, opCount,
lastCodeSepLocation, opcode, verifyFlags);
                break;
            case OP_CHECKLOCKTIMEVERIFY:
                if (!verifyFlags.contains(VerifyFlag.CHECKLOCKTIMEVERIFY)) {
                    // not enabled; treat as a NOP2
                    if (verifyFlags.contains(VerifyFlag.DISCOURAGE_UPGRADABLE_NOPS)) {
                        throw new ScriptException("Script used a reserved opcode " + opcode);
                    }
                    break;
                }
                executeCheckLockTimeVerify(txContainingThis, (int) index, script, stack,
lastCodeSepLocation, opcode, verifyFlags);
```

```
                    break;
                case OP_NOP1:
                case OP_NOP3:
                case OP_NOP4:
                case OP_NOP5:
                case OP_NOP6:
                case OP_NOP7:
                case OP_NOP8:
                case OP_NOP9:
                case OP_NOP10:
                    if (verifyFlags.contains(VerifyFlag.DISCOURAGE_UPGRADABLE_NOPS)) {
                        throw new ScriptException("Script used a reserved opcode " + opcode);
                    }
                    break;

                default:
                    throw new ScriptException("Script used a reserved opcode " + opcode);
            }
        }

        if (stack.size() + altstack.size() > 1000 || stack.size() + altstack.size() < 0) {
            throw new ScriptException("Stack size exceeded range");
        }
    }

    if (!ifStack.isEmpty()) {
        throw new ScriptException("OP_IF/OP_NOTIF without OP_ENDIF");
    }
}

// This is more or less a direct translation of the code in Bitcoin Core
private static void executeCheckLockTimeVerify(Transaction txContainingThis, int index, Script
script, LinkedList<byte[]> stack,
                                               int lastCodeSepLocation, int opcode,
                                               Set<VerifyFlag> verifyFlags) throws ScriptException {
    if (stack.size() < 1) {
        throw new ScriptException("Attempted OP_CHECKLOCKTIMEVERIFY on a stack with
size < 1");
    }

    // Thus as a special case we tell CScriptNum to accept up
    // to 5-byte bignums to avoid year 2038 issue.
```

```
        final BigInteger nLockTime = castToBigInteger(stack.getLast(), 5);

        if (nLockTime.compareTo(BigInteger.ZERO) < 0) {
            throw new ScriptException("Negative locktime");
        }

        // There are two kinds of nLockTime, need to ensure we're comparing apples-to-apples
        /*if (!(
            ((txContainingThis.getLockTime() <  Transaction.LOCKTIME_THRESHOLD) &&
(nLockTime.compareTo(Transaction.LOCKTIME_THRESHOLD_BIG)) < 0) ||
            ((txContainingThis.getLockTime() >= Transaction.LOCKTIME_THRESHOLD) &&
(nLockTime.compareTo(Transaction.LOCKTIME_THRESHOLD_BIG)) >= 0))
        )*/
        throw new ScriptException("Locktime requirement type mismatch");

        // Now that we know we're comparing apples-to-apples, the
        // comparison is a simple numeric one.
        /*if (nLockTime.compareTo(BigInteger.valueOf(txContainingThis.getLockTime())) > 0)
            throw new ScriptException("Locktime requirement not satisfied");*/

        // Finally the nLockTime feature can be disabled and thus
        // CHECKLOCKTIMEVERIFY bypassed if every txin has been
        // finalized by setting nSequence to maxint. The
        // transaction would be allowed into the blockchain, making
        // the opcode ineffective.
        //
        // Testing if this vin is not final is sufficient to
        // prevent this condition. Alternatively we could test all
        // inputs, but testing just this input minimizes the data
        // required to prove correct CHECKLOCKTIMEVERIFY execution.
        /*if (!txContainingThis.getInput(index).hasSequence())
            throw new ScriptException("Transaction contains a final transaction input for a
CHECKLOCKTIMEVERIFY script.");*/
    }

    private static void executeCheckSig(Transaction txContainingThis, int index, Script script,
LinkedList<byte[]> stack,
                                        int lastCodeSepLocation, int opcode,
                                        Set<VerifyFlag> verifyFlags) throws ScriptException {
        final boolean requireCanonical = verifyFlags.contains(VerifyFlag.STRICTENC)
                || verifyFlags.contains(VerifyFlag.DERSIG)
                || verifyFlags.contains(VerifyFlag.LOW_S);
```

```
        if (stack.size() < 2) {
            throw new ScriptException("Attempted OP_CHECKSIG(VERIFY) on a stack with size <
2");
        }
        byte[] pubKey = stack.pollLast();
        byte[] sigBytes = stack.pollLast();

        byte[] prog = script.getProgram();
        byte[] connectedScript = Arrays.copyOfRange(prog, lastCodeSepLocation, prog.length);

        UnsafeByteArrayOutputStream outStream = new
UnsafeByteArrayOutputStream(sigBytes.length + 1);
        try {
            writeBytes(outStream, sigBytes);
        } catch (IOException e) {
            throw new RuntimeException(e); // Cannot happen
        }
        connectedScript = removeAllInstancesOf(connectedScript, outStream.toByteArray());

        // TODO: Use int for indexes everywhere, we can't have that many inputs/outputs
        boolean sigValid = false;
        try {
            sigValid = verifySign(txContainingThis.getHash().getDigestBytes(), sigBytes, pubKey);

            /*ransactionSignature sig  = TransactionSignature.decodeFromBitcoin(sigBytes,
requireCanonical,
                    verifyFlags.contains(VerifyFlag.LOW_S));

            // TODO: Should check hash type is known
            Sha256Hash hash = txContainingThis.hashForSignature(index, connectedScript, (byte)
sig.sighashFlags);
            sigValid = ECKey.verify(hash.getBytes(), sig, pubKey);*/
        } catch (Exception e1) {
            // There is (at least) one exception that could be hit here (EOFException, if the sig is too
short)
            // Because I can't verify there aren't more, we use a very generic Exception catch

            // This RuntimeException occurs when signing as we run partial/invalid scripts to see if
they need more
            // signing work to be done inside LocalTransactionSigner.signInputs.
            if (!e1.getMessage().contains("Reached past end of ASN.1 stream")) {
                log.warn("Signature checking failed!", e1);
```

```
            }
        }
        //System.out.println(opcode == OP_CHECKSIG);
        if (opcode == OP_CHECKSIG) {
            stack.add(sigValid ? new byte[]{1} : new byte[]{});
        } else if (opcode == OP_CHECKSIGVERIFY) {
            if (!sigValid) {
                throw new ScriptException("Script failed OP_CHECKSIGVERIFY");
            }
        }
    }

    private static int executeMultiSig(Transaction txContainingThis, int index, Script script,
LinkedList<byte[]> stack,
                                       int opCount, int lastCodeSepLocation, int opcode,
                                       Set<VerifyFlag> verifyFlags) throws ScriptException {
        final boolean requireCanonical = verifyFlags.contains(VerifyFlag.STRICTENC)
                || verifyFlags.contains(VerifyFlag.DERSIG)
                || verifyFlags.contains(VerifyFlag.LOW_S);
        if (stack.size() < 2) {
            throw new ScriptException("Attempted OP_CHECKMULTISIG(VERIFY) on a stack with
size < 2");
        }
        int pubKeyCount = castToBigInteger(stack.pollLast()).intValue();
        if (pubKeyCount < 0 || pubKeyCount > 20) {
            throw new ScriptException("OP_CHECKMULTISIG(VERIFY) with pubkey count out of
range");
        }
        opCount += pubKeyCount;
        if (opCount > 201) {
            throw new ScriptException("Total op count > 201 during
OP_CHECKMULTISIG(VERIFY)");
        }
        if (stack.size() < pubKeyCount + 1) {
            throw new ScriptException("Attempted OP_CHECKMULTISIG(VERIFY) on a stack with
size < num_of_pubkeys + 2");
        }

        LinkedList<byte[]> pubkeys = new LinkedList<byte[]>();
        for (int i = 0; i < pubKeyCount; i++) {
            byte[] pubKey = stack.pollLast();
            pubkeys.add(pubKey);
```

```
        }

        int sigCount = castToBigInteger(stack.pollLast()).intValue();
        if (sigCount < 0 || sigCount > pubKeyCount) {
            throw new ScriptException("OP_CHECKMULTISIG(VERIFY) with sig count out of range");
        }
        if (stack.size() < sigCount + 1) {
            throw new ScriptException("Attempted OP_CHECKMULTISIG(VERIFY) on a stack with
size < num_of_pubkeys + num_of_signatures + 3");
        }

        LinkedList<byte[]> sigs = new LinkedList<byte[]>();
        for (int i = 0; i < sigCount; i++) {
            byte[] sig = stack.pollLast();
            sigs.add(sig);
        }

        byte[] prog = script.getProgram();
        byte[] connectedScript = Arrays.copyOfRange(prog, lastCodeSepLocation, prog.length);

        for (byte[] sig : sigs) {
            UnsafeByteArrayOutputStream outStream = new
UnsafeByteArrayOutputStream(sig.length + 1);
            try {
                writeBytes(outStream, sig);
            } catch (IOException e) {
                throw new RuntimeException(e); // Cannot happen
            }
            connectedScript = removeAllInstancesOf(connectedScript, outStream.toByteArray());
        }

        boolean valid = true;
        while (sigs.size() > 0) {
            byte[] pubKey = pubkeys.pollFirst();
            // We could reasonably move this out of the loop, but because signature verification is
significantly
            // more expensive than hashing, its not a big deal.
            try {
                if (ECKey.verify(txContainingThis.getHash().getDigestBytes(), sigs.getFirst(), pubKey)) {
                    sigs.pollFirst();
                }
               /* TransactionSignature sig = TransactionSignature.decodeFromBitcoin(sigs.getFirst(),
```

```
requireCanonical);
            Sha256Hash hash = txContainingThis.hashForSignature(index, connectedScript, (byte)
sig.sighashFlags);
            if (ECKey.verify(hash.getBytes(), sig, pubKey))
                sigs.pollFirst();*/
            } catch (Exception e) {
                // There is (at least) one exception that could be hit here (EOFException, if the sig is too
short)
                // Because I can't verify there aren't more, we use a very generic Exception catch
                e.printStackTrace();
            }
            if (sigs.size() > pubkeys.size()) {
                valid = false;
                break;
            }
        }
        // We uselessly remove a stack object to emulate a Bitcoin Core bug.
        byte[] nullDummy = stack.pollLast();
        if (verifyFlags.contains(VerifyFlag.NULLDUMMY) && nullDummy.length > 0) {
            throw new ScriptException("OP_CHECKMULTISIG(VERIFY) with non-null nulldummy: " +
Arrays.toString(nullDummy));
        }

        if (opcode == OP_CHECKMULTISIG) {
            stack.add(valid ? new byte[]{1} : new byte[]{});
        } else if (opcode == OP_CHECKMULTISIGVERIFY) {
            if (!valid) {
                throw new ScriptException("Script failed OP_CHECKMULTISIGVERIFY");
            }
        }
        return opCount;
    }

    /**
     * Verifies that this script (interpreted as a scriptSig) correctly spends the given scriptPubKey,
enabling all
     * validation rules.scriptPubKey
     *
     * @param txContainingThis The transaction in which this input scriptSig resides.
     *                         Accessing txContainingThis from another thread while this method runs results
in undefined behavior.
     * @param scriptSigIndex   The index in txContainingThis of the scriptSig (note: NOT the index
```

```
of the scriptPubKey).
     * @param scriptPubKey     The connected scriptPubKey containing the conditions needed to
claim the value.
     *                         instead so that verification flags do not change as new verification options
     *                         are added.
     *                          txContainingThis+scriptSigIndexInput
     */
    @Deprecated
    public boolean correctlySpends(Transaction txContainingThis, long scriptSigIndex, Script
scriptPubKey) {
        return correctlySpends(txContainingThis, scriptSigIndex, scriptPubKey,
ALL_VERIFY_FLAGS);
    }

    public boolean correctlyNulsSpends(Transaction txContainingThis, long scriptSigIndex, Script
scriptPubKey) {
        return correctlySpends(txContainingThis, scriptSigIndex, scriptPubKey,
ALL_VERIFY_FLAGS);
    }

    /**
     * Verifies that this script (interpreted as a scriptSig) correctly spends the given scriptPubKey.
     *
     * @param txContainingThis  The transaction in which this input scriptSig resides.
     *                         Accessing txContainingThis from another thread while this method runs results
in undefined behavior.
     * @param scriptSigIndex   scriptSig The index in txContainingThis of the scriptSig (note: NOT
the index of the scriptPubKey).
     * @param scriptPubKey     scriptPubKey   The connected scriptPubKey containing the
conditions needed to claim the value.
     * @param verifyFlags       Each flag enables one validation rule. If in doubt, use {@link
#correctlySpends(Transaction, long, Script)}
     *                         which sets all flags.
     *                         <p>
     *                         scriptSigscriptPubKey
     */
    public boolean correctlySpends(Transaction txContainingThis, long scriptSigIndex, Script
scriptPubKey, Set<VerifyFlag> verifyFlags) {
        try {
            if (verifyFlags == null)
            {
                verifyFlags = ALL_VERIFY_FLAGS;
```

```
        }
        if (getProgram().length > 10000 || scriptPubKey.getProgram().length > 10000) {
            throw new ScriptException("Script larger than 10,000 bytes");
        }

        LinkedList<byte[]> stack = new LinkedList<byte[]>();
        LinkedList<byte[]> p2shStack = null;

        executeScript(txContainingThis, scriptSigIndex, this, stack, verifyFlags);
        if (verifyFlags.contains(VerifyFlag.P2SH)) {
            p2shStack = new LinkedList<byte[]>(stack);
        }
        executeScript(txContainingThis, scriptSigIndex, scriptPubKey, stack, verifyFlags);
        if (stack.size() == 0) {
            throw new ScriptException("Stack empty at end of script execution.");
        }

        if (!castToBool(stack.pollLast())) {
            throw new ScriptException("Script resulted in a non-true stack: " + stack);
        }
        // TODO: Check if we can take out enforceP2SH if there's a checkpoint at the enforcement
block.
        if (verifyFlags.contains(VerifyFlag.P2SH) && scriptPubKey.isPayToScriptHash()) {
            for (ScriptChunk chunk : chunks) {
                if (chunk.isOpCode() && chunk.opcode > OP_16) {
                    throw new ScriptException("Attempted to spend a P2SH scriptPubKey with a script
that contained script ops");
                }
            }

            byte[] scriptPubKeyBytes = p2shStack.pollLast();
            Script scriptPubKeyP2SH = new Script(scriptPubKeyBytes);

            executeScript(txContainingThis, scriptSigIndex, scriptPubKeyP2SH, p2shStack,
verifyFlags);

            if (p2shStack.size() == 0) {
                throw new ScriptException("P2SH stack empty at end of script execution.");
            }

            if (!castToBool(p2shStack.pollLast())) {
                throw new ScriptException("P2SH script execution resulted in a non-true stack");
```

```
            }
        }
    } catch (ScriptException e) {
        return false;
    }


    // P2SH is pay to script hash. It means that the scriptPubKey has a special form which is a valid
    // program but it has "useless" form that if evaluated as a normal program always returns true.
    // Instead, miners recognize it as special based on its template - it provides a hash of the real scriptPubKey
    // and that must be provided by the input. The goal of this bizarre arrangement is twofold:
    //
    // (1) You can sum up a large, complex script (like a CHECKMULTISIG script) with an address that's the same
    //     size as a regular address. This means it doesn't overload scannable QR codes/NFC tags or become
    //     un-wieldy to copy/paste.
    // (2) It allows the working set to be smaller: nodes perform best when they can store as many unspent outputs
    //     in RAM as possible, so if the outputs are made smaller and the inputs get bigger, then it's better for
    //     overall scalability and performance.
    return true;
}

// Utility that doesn't copy for internal use
private byte[] getQuickProgram() {
    if (program != null) {
        return program;
    }
    return getProgram();
}

/**
 * @return The script type.
 */
public ScriptType getScriptType() {
    ScriptType type = ScriptType.NO_TYPE;
    if (isSentToAddress()) {
```

```
            type = ScriptType.P2PKH;
        } else if (isSentToRawPubKey()) {
            type = ScriptType.PUB_KEY;
        } else if (isPayToScriptHash()) {
            type = ScriptType.P2SH;
        }
        return type;
    }


    /**
     *
     */
    public static boolean verifySign(byte[] digestData, byte[] signData, byte[] publicKey) {
        return ECKey.verify(digestData, signData, publicKey);
    }

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

    @Override
    public int hashCode() {
        return Arrays.hashCode(getQuickProgram());
    }
}

60:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\ScriptBuilder.java
import io.nuls.kernel.model.Address;
import io.nuls.kernel.utils.SerializeUtils;

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

```
import static com.google.common.base.Preconditions.checkArgument;
import static com.google.common.base.Preconditions.checkState;
import static io.nuls.kernel.script.ScriptOpCodes.*;

/**
 * <p>Tools for the construction of commonly used script types. You don't normally need this as it's hidden behind
 * convenience methods on {@link io.nuls.kernel.model}, but they are useful when working with the
 * protocol at a lower level.</p>
 */
public class ScriptBuilder {
    private List<ScriptChunk> chunks;               //

    /**
     * Creates a fresh ScriptBuilder with an empty program.
     */
    public ScriptBuilder() {
        chunks = Lists.newLinkedList();
    }

    /**
     * Creates a fresh ScriptBuilder with the given program as the starting point.
     */
    public ScriptBuilder(Script template) {
        chunks = new ArrayList<ScriptChunk>(template.getChunks());
    }

    /**
     * Adds the given chunk to the end of the program
     */
    public ScriptBuilder addChunk(ScriptChunk chunk) {
        return addChunk(chunks.size(), chunk);
    }

    /**
     * Adds the given chunk at the given index in the program
     *
     */
    public ScriptBuilder addChunk(int index, ScriptChunk chunk) {
        chunks.add(index, chunk);
```

```
        return this;
    }

    /**
     * Adds the given opcode to the end of the program.
     *
     */
    public ScriptBuilder op(int opcode) {
        return op(chunks.size(), opcode);
    }

    /**
     * Adds the given opcode to the given index in the program
     *
     */
    public ScriptBuilder op(int index, int opcode) {
        checkArgument(opcode > OP_PUSHDATA4);
        return addChunk(index, new ScriptChunk(opcode, null));
    }

    /**
     * Adds a copy of the given byte array as a data element (i.e. PUSHDATA) at the end of the program.
     *
     */
    public ScriptBuilder data(byte[] data) {
        if (data.length == 0) {
            return smallNum(0);
        } else {
            return data(chunks.size(), data);
        }
    }

    /**
     * Adds a copy of the given byte array as a data element (i.e. PUSHDATA) at the given index in the program.
     *
     */
    public ScriptBuilder data(int index, byte[] data) {
        // implements BIP62
        byte[] copy = Arrays.copyOf(data, data.length);
        int opcode;
```

```
    /**
     * 0OP_0
     * */
    if (data.length == 0) {
        opcode = OP_0;
    }
    /**
     * 1
     * */
    else if (data.length == 1) {
        byte b = data[0];
        /**
         * 1116
         * */
        if (b >= 1 && b <= 16) {
            opcode = Script.encodeToOpN(b);
        } else {
            opcode = 1;
        }
    }
    /**
     * 0x4c
     * */
    else if (data.length < OP_PUSHDATA1) {
        opcode = data.length;
    }
    /**
     * 256OP_PUSHDATA10x4c
     * */
    else if (data.length < 256) {
        opcode = OP_PUSHDATA1;
    }
    /**
     * 65536OP_PUSHDATA20x4d
     * */
    else if (data.length < 65536) {
        opcode = OP_PUSHDATA2;
    } else {
        throw new RuntimeException("Unimplemented");
    }
    return addChunk(index, new ScriptChunk(opcode, copy));
}
```

```
/**
 * Adds the given number to the end of the program. Automatically uses
 * shortest encoding possible.
 */
public ScriptBuilder number(long num) {
    return number(chunks.size(), num);
}

/**
 * Adds the given number to the given index in the program. Automatically
 * uses shortest encoding possible.
 * long
 */
public ScriptBuilder number(int index, long num) {
    if (num == -1) {
        return op(index, OP_1NEGATE);
    } else if (num >= 0 && num <= 16) {
        return addChunk(index, new ScriptChunk(Script.encodeToOpN((int) num), null));
    } else {
        return bigNum(index, num);
    }
}

/**
 * Adds the given number as a OP_N opcode to the end of the program.
 * Only handles values 0-16 inclusive.
 *
 * @see #(int)
 */
public ScriptBuilder smallNum(int num) {
    return smallNum(chunks.size(), num);
}

/**
 * Adds the given number as a push data chunk.
 * This is intended to use for negative numbers or values > 16, and although
 * it will accept numbers in the range 0-16 inclusive, the encoding would be
 * considered non-standard.
 *
 * @see #(int)
 */
```

```
protected ScriptBuilder bigNum(long num) {
    return bigNum(chunks.size(), num);
}

/**
 * Adds the given number as a OP_N opcode to the given index in the program.
 * Only handles values 0-16 inclusive.
 *
 * @see #(int)
 */
public ScriptBuilder smallNum(int index, int num) {
    checkArgument(num >= 0, "Cannot encode negative numbers with smallNum");
    checkArgument(num <= 16, "Cannot encode numbers larger than 16 with smallNum");
    return addChunk(index, new ScriptChunk(Script.encodeToOpN(num), null));
}

/**
 * Adds the given number as a push data chunk to the given index in the program.
 * This is intended to use for negative numbers or values > 16, and although
 * it will accept numbers in the range 0-16 inclusive, the encoding would be
 * considered non-standard.
 *
 *
 * @see #(int)
 */
protected ScriptBuilder bigNum(int index, long num) {
    final byte[] data;

    if (num == 0) {
        data = new byte[0];
    } else {
        Stack<Byte> result = new Stack<Byte>();
        final boolean neg = num < 0;
        long absvalue = Math.abs(num);

        while (absvalue != 0) {
            result.push((byte) (absvalue & 0xff));
            absvalue >>= 8;
        }

        if ((result.peek() & 0x80) != 0) {
            // The most significant byte is >= 0x80, so push an extra byte that
```

```
            // contains just the sign of the value.
            result.push((byte) (neg ? 0x80 : 0));
        } else if (neg) {
            // The most significant byte is < 0x80 and the value is negative,
            // set the sign bit so it is subtracted and interpreted as a
            // negative when converting back to an integral.
            result.push((byte) (result.pop() | 0x80));
        }

        data = new byte[result.size()];
        for (int byteIdx = 0; byteIdx < data.length; byteIdx++) {
            data[byteIdx] = result.get(byteIdx);
        }
    }

    // At most the encoded value could take up to 8 bytes, so we don't need
    // to use OP_PUSHDATA opcodes
    return addChunk(index, new ScriptChunk(data.length, data));
}

/**
 * Creates a new immutable Script based on the state of the builder.
 *
 */
public Script build() {
    return new Script(chunks);
}

/**
 * Creates a scriptPubKey that encodes payment to the given address.
 * OutputScript/scriptPublicKry
 */
public static Script createOutputScript(byte[] address, int type) {
    //P2SHP2SH
    if (type == 0) {
        // OP_HASH160 <scriptHash> OP_EQUAL
        return new ScriptBuilder()
                .op(OP_HASH160)
                .data(address)
                .op(OP_EQUAL)
                .build();
    } else {
```

```
            // OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
            return new ScriptBuilder()
                    .op(OP_DUP)
                    .op(OP_HASH160)
                    .data(address)
                    .op(OP_EQUALVERIFY)
                    .op(OP_CHECKSIG)
                    .build();
        }
    }

    /**
     * Creates a scriptPubKey that encodes payment to the given raw public key.
     * OutputScript/scriptPublicKry
     * P2PKPay-to-Public-Key
     */
    public static Script createOutputScript(ECKey key) {
        return new ScriptBuilder().data(key.getPubKey()).op(OP_CHECKSIG).build();
    }

    /**
     * Creates a scriptSig that can redeem a pay-to-address output.
     * If given signature is null, incomplete scriptSig will be created with OP_0 instead of signature
     * pay-to-addressinputScript/scriptSigOutputScript/scriptPublicKry
     */
    public static Script createInputScript(@Nullable TransactionSignature signature, ECKey
pubKey) {
        byte[] pubkeyBytes = pubKey.getPubKey();
        //byte[] sigBytes = signature != null ? signature.encodeToBitcoin() : new byte[]{};
        byte[] sigBytes = null;
        return new ScriptBuilder().data(sigBytes).data(pubkeyBytes).build();
    }

    /**
     * Creates a scriptSig that can redeem a pay-to-address output.
     * If given signature is null, incomplete scriptSig will be created with OP_0 instead of signature
     * pay-to-addressinputScript/scriptSigOutputScript/scriptPublicKry
     */
    public static Script createNulsInputScript(@Nullable byte[] signBytes, byte[] pubKeyBytes) {
        return new ScriptBuilder().data(signBytes).data(pubKeyBytes).build();
    }
```

```
    /**
     * Creates a scriptSig that can redeem a pay-to-pubkey output.
     * If given signature is null, incomplete scriptSig will be created with OP_0 instead of signature
     * pay-to-public_keyinputScript/scriptSigOutputScript/scriptPublicKry
     */
    public static Script createInputScript(@Nullable TransactionSignature signature) {
        //byte[] sigBytes = signature != null ? signature.encodeToBitcoin() : new byte[]{};
        byte[] sigBytes = null;
        return new ScriptBuilder().data(sigBytes).build();
    }

    /**
     * Creates a program that requires at least N of the given keys to sign, using
OP_CHECKMULTISIG.
     * OutputScript/scriptPublicKry
     */
    public static Script createMultiSigOutputScript(int threshold, List<ECKey> pubkeys) {
        checkArgument(threshold > 0);
        checkArgument(threshold <= pubkeys.size());
        checkArgument(pubkeys.size() <= 16);  // That's the max we can represent with a single
opcode.
        ScriptBuilder builder = new ScriptBuilder();
        builder.smallNum(threshold);
        for (ECKey key : pubkeys) {
            builder.data(key.getPubKey());
        }
        builder.smallNum(pubkeys.size());
        builder.op(OP_CHECKMULTISIG);
        return builder.build();
    }

    /**
     * Creates a program that requires at least N of the given keys to sign, using
OP_CHECKMULTISIG.
     * OutputScript/scriptPublicKry
     */
    public static Script createNulsMultiSigOutputScript(int threshold, List<String> pubkeys) {
        checkArgument(threshold > 0);
        checkArgument(threshold <= pubkeys.size());
        checkArgument(pubkeys.size() <= 16);  // That's the max we can represent with a single
opcode.
        ScriptBuilder builder = new ScriptBuilder();
```

```
        builder.smallNum(threshold);
        for (String pubKey : pubkeys) {
            builder.data(Hex.decode(pubKey));
        }
        builder.smallNum(pubkeys.size());
        builder.op(OP_CHECKMULTISIG);
        return builder.build();
    }

    /**
     * Create a program that satisfies an OP_CHECKMULTISIG program.
     * inputScript/scriptSig
     **/
    public static Script createByteNulsMultiSigOutputScript(int threshold, List<byte[]> pubkeys) {
        checkArgument(threshold > 0);
        checkArgument(threshold <= pubkeys.size());
        checkArgument(pubkeys.size() <= 16);  // That's the max we can represent with a single
opcode.
        ScriptBuilder builder = new ScriptBuilder();
        builder.smallNum(threshold);
        for (byte[] pubkey : pubkeys) {
            builder.data(pubkey);
        }
        builder.smallNum(pubkeys.size());
        builder.op(OP_CHECKMULTISIG);
        return builder.build();
    }

    /**
     * Create a program that satisfies an OP_CHECKMULTISIG program.
     * inputScript/scriptSig
     **/
    public static Script createMultiSigInputScript(List<TransactionSignature> signatures) {
        List<byte[]> sigs = new ArrayList<byte[]>(signatures.size());
        for (TransactionSignature signature : signatures) {
            //sigs.add(signature.encodeToBitcoin());
        }
        return createMultiSigInputScriptBytes(sigs, null);
    }

    /**
     * Create a program that satisfies an OP_CHECKMULTISIG program.
```

```
     */
    public static Script createMultiSigInputScript(TransactionSignature... signatures) {
        return createMultiSigInputScript(Arrays.asList(signatures));
    }

    /**
     * Create a program that satisfies an OP_CHECKMULTISIG program, using pre-encoded
signatures.
     */
    public static Script createMultiSigInputScriptBytes(List<byte[]> signatures) {
        return createMultiSigInputScriptBytes(signatures, null);
    }

    /**
     * Create a program that satisfies a pay-to-script hashed OP_CHECKMULTISIG program.
     * If given signature list is null, incomplete scriptSig will be created with OP_0 instead of
signatures
     * P2SH
     */
    public static Script createP2SHMultiSigInputScript(@Nullable List<TransactionSignature>
signatures,
                                                       Script multisigProgram) {
        List<byte[]> sigs = new ArrayList<byte[]>();
        if (signatures == null) {
            // create correct number of empty signatures
            int numSigs = multisigProgram.getNumberOfSignaturesRequiredToSpend();  //UTXO
            for (int i = 0; i < numSigs; i++) {
                sigs.add(new byte[]{});
            }
        } else {
            for (TransactionSignature signature : signatures) {
                //sigs.add(signature.encodeToBitcoin());
            }
        }
        return createMultiSigInputScriptBytes(sigs, multisigProgram.getProgram());
    }

    /**
     * Create a program that satisfies a pay-to-script hashed OP_CHECKMULTISIG program.
     * If given signature list is null, incomplete scriptSig will be created with OP_0 instead of
signatures
     * P2SH
```

```
     */
    public static Script createNulsP2SHMultiSigInputScript(@Nullable List<byte[]> signatures,
                                                           Script multisigProgram) {
        List<byte[]> sigs = new ArrayList<byte[]>();
        if (signatures == null) {
            // create correct number of empty signatures
            int numSigs = multisigProgram.getNumberOfSignaturesRequiredToSpend();  //UTXO
            for (int i = 0; i < numSigs; i++) {
                sigs.add(new byte[]{});
            }
        } else {
            for (byte[] signature : signatures) {
                sigs.add(signature);
            }
        }
        return createMultiSigInputScriptBytes(sigs, multisigProgram.getProgram());
    }

    /**
     * Create a program that satisfies an OP_CHECKMULTISIG program, using pre-encoded signatures.
     * Optionally, appends the script program bytes if spending a P2SH output.
     */
    public static Script createMultiSigInputScriptBytes(List<byte[]> signatures, @Nullable byte[] multisigProgramBytes) {
        checkArgument(signatures.size() <= 16);
        ScriptBuilder builder = new ScriptBuilder();
        builder.smallNum(0);  // Work around a bug in CHECKMULTISIG that is now a required part of the protocol.
        for (byte[] signature : signatures) {
            builder.data(signature);
        }
        if (multisigProgramBytes != null) {
            builder.data(multisigProgramBytes);
        }
        return builder.build();
    }


    /**
     * Returns a copy of the given scriptSig with the signature inserted in the given position.
     * <p>
```

```
    * This function assumes that any missing sigs have OP_0 placeholders. If given scriptSig
already has all the signatures
    * in place, IllegalArgumentException will be thrown.
    *
    * @param targetIndex     where to insert the signature
    * @param sigsPrefixCount how many items to copy verbatim (e.g. initial OP_0 for multisig)
    * @param sigsSuffixCount how many items to copy verbatim at end (e.g. redeemScript for
P2SH)
    */
   public static Script updateScriptWithSignature(Script scriptSig, byte[] signature, int targetIndex,
                                                  int sigsPrefixCount, int sigsSuffixCount) {
       ScriptBuilder builder = new ScriptBuilder();
       List<ScriptChunk> inputChunks = scriptSig.getChunks();
       int totalChunks = inputChunks.size();

       // Check if we have a place to insert, otherwise just return given scriptSig unchanged.
       // We assume here that OP_0 placeholders always go after the sigs, so
       // to find if we have sigs missing, we can just check the chunk in latest sig position
       boolean hasMissingSigs = inputChunks.get(totalChunks - sigsSuffixCount -
1).equalsOpCode(OP_0);
       checkArgument(hasMissingSigs, "ScriptSig is already filled with signatures");

       // copy the prefix
       for (ScriptChunk chunk : inputChunks.subList(0, sigsPrefixCount)) {
           builder.addChunk(chunk);
       }

       // copy the sigs
       int pos = 0;
       boolean inserted = false;
       for (ScriptChunk chunk : inputChunks.subList(sigsPrefixCount, totalChunks -
sigsSuffixCount)) {
           if (pos == targetIndex) {
               inserted = true;
               builder.data(signature);
               pos++;
           }
           if (!chunk.equalsOpCode(OP_0)) {
               builder.addChunk(chunk);
               pos++;
           }
       }
```

```
        // add OP_0's if needed, since we skipped them in the previous loop
        while (pos < totalChunks - sigsPrefixCount - sigsSuffixCount) {
            if (pos == targetIndex) {
                inserted = true;
                builder.data(signature);
            } else {
                builder.addChunk(new ScriptChunk(OP_0, null));
            }
            pos++;
        }

        // copy the suffix
        for (ScriptChunk chunk : inputChunks.subList(totalChunks - sigsSuffixCount, totalChunks)) {
            builder.addChunk(chunk);
        }

        checkState(inserted);
        return builder.build();
    }

    /**
     * Creates a scriptPubKey that sends to the given script hash. Read
     * <a href="https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki">BIP 16</a> to learn more about this
     * kind of script.
     * hashP2SH
     */
    public static Script createP2SHOutputScript(byte[] hash) {
        checkArgument(hash.length == 23);
        return new ScriptBuilder().op(OP_HASH160).data(hash).op(OP_EQUAL).build();
    }

    /**
     * Creates a scriptPubKey for the given redeem script.
     * P2SH
     */
    public static Script createP2SHOutputScript(Script redeemScript) {
        Address address = new Address(NulsContext.DEFAULT_CHAIN_ID, NulsContext.P2SH_ADDRESS_TYPE, SerializeUtils.sha256hash160(redeemScript.getProgram()));
        //byte[] hash = Utils.sha256hash160(redeemScript.getProgram());
```

```
        byte[] hash = address.getAddressBytes();
        return ScriptBuilder.createP2SHOutputScript(hash);
    }

    /**
     * Creates a P2SH output script with given public keys and threshold. Given public keys will be placed in
     * redeem script in the lexicographical sorting order.
     * <p>
     * P2SH
     */
    public static Script createP2SHOutputScript(int threshold, List<ECKey> pubkeys) {
        Script redeemScript = createRedeemScript(threshold, pubkeys);
        return createP2SHOutputScript(redeemScript);
    }

    /**
     * Creates redeem script with given public keys and threshold. Given public keys will be placed in
     * redeem script in the lexicographical sorting order.
     */
    public static Script createRedeemScript(int threshold, List<ECKey> pubkeys) {
        pubkeys = new ArrayList<ECKey>(pubkeys);
        Collections.sort(pubkeys, ECKey.PUBKEY_COMPARATOR);
        return ScriptBuilder.createMultiSigOutputScript(threshold, pubkeys);
    }

    /**
     * Creates redeem script with given public keys and threshold. Given public keys will be placed in
     * redeem script in the lexicographical sorting order.
     */
    public static Script createNulsRedeemScript(int threshold, List<String> pubkeys) {
        pubkeys = new ArrayList<String>(pubkeys);
        Collections.sort(pubkeys, PUBKEY_COMPARATOR);
        return ScriptBuilder.createNulsMultiSigOutputScript(threshold, pubkeys);
    }

    /**
     * Creates redeem script with given public keys and threshold. Given public keys will be placed in
     * redeem script in the lexicographical sorting order.
```

```
     */
    public static Script createByteNulsRedeemScript(int threshold, List<byte[]> pubkeys) {
        pubkeys = new ArrayList<byte[]>(pubkeys);
        Collections.sort(pubkeys, PUBKEY_BYTE_COMPARATOR);
        return ScriptBuilder.createByteNulsMultiSigOutputScript(threshold, pubkeys);
    }

    /**
     * Creates a script of the form OP_RETURN [data]. This feature allows you to attach a small piece of data (like
     * a hash of something stored elsewhere) to a zero valued output which can never be spent and thus does not pollute
     * the ledger.
     */
    public static Script createOpReturnScript(byte[] data) {
        checkArgument(data.length <= 80);
        return new ScriptBuilder().op(OP_RETURN).data(data).build();
    }

    public static final Comparator<String> PUBKEY_COMPARATOR = new Comparator<String>() {
        private Comparator<byte[]> comparator = UnsignedBytes.lexicographicalComparator();

        @Override
        public int compare(String k1, String k2) {
            return comparator.compare(Hex.decode(k1), Hex.decode(k2));
        }
    };

    public static final Comparator<byte[]> PUBKEY_BYTE_COMPARATOR = new Comparator<byte[]>() {
        private Comparator<byte[]> comparator = UnsignedBytes.lexicographicalComparator();

        @Override
        public int compare(byte[] k1, byte[] k2) {
            return comparator.compare(k1, k2);
        }
    };
}
```

61:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\script\ScriptChunk.java

```
import javax.annotation.Nullable;
```

```
import java.io.IOException;
import java.io.OutputStream;
import java.util.Arrays;

import static com.google.common.base.Preconditions.checkState;
import static io.nuls.kernel.script.ScriptOpCodes.*;

/**
 * A script element that is either a data push (signature, pubkey, etc) or a non-push (logic, numeric, etc) operation.
 */
public class ScriptChunk {
    /**
     * Operation to be executed. Opcodes are defined in {@link ScriptOpCodes}.
     */
    public final int opcode;
    /**
     * For push operations, this is the vector to be pushed on the stack. For {@link ScriptOpCodes#OP_0}, the vector is
     * empty. Null for non-push operations.
     */
    @Nullable
    public final byte[] data;
    private int startLocationInProgram;

    public ScriptChunk(int opcode, byte[] data) {
        this(opcode, data, -1);
    }

    public ScriptChunk(int opcode, byte[] data, int startLocationInProgram) {
        this.opcode = opcode;
        this.data = data;
        this.startLocationInProgram = startLocationInProgram;
    }

    public boolean equalsOpCode(int opcode) {
        return opcode == this.opcode;
    }

    /**
     * If this chunk is a single byte of non-pushdata content (could be OP_RESERVED or some invalid Opcode)
```

```
     */
    public boolean isOpCode() {
        return opcode > OP_PUSHDATA4;
    }

    /**
     * Returns true if this chunk is pushdata content, including the single-byte pushdatas.
     */
    public boolean isPushData() {
        return opcode <= OP_16;
    }

    public int getStartLocationInProgram() {
        checkState(startLocationInProgram >= 0);
        return startLocationInProgram;
    }

    /**
     * If this chunk is an OP_N opcode returns the equivalent integer value.
     */
    public int decodeOpN() {
        checkState(isOpCode());
        return Script.decodeFromOpN(opcode);
    }

    /**
     * Called on a pushdata chunk, returns true if it uses the smallest possible way (according to
BIP62) to push the data.
     */
    public boolean isShortestPossiblePushData() {
        checkState(isPushData());
        if (data == null) {
            return true;   // OP_N
        }
        if (data.length == 0) {
            return opcode == OP_0;
        }
        if (data.length == 1) {
            byte b = data[0];
            if (b >= 0x01 && b <= 0x10) {
                return opcode == OP_1 + b - 1;
            }
```

```
            if ((b & 0xFF) == 0x81) {
                return opcode == OP_1NEGATE;
            }
        }
        if (data.length < OP_PUSHDATA1) {
            return opcode == data.length;
        }
        if (data.length < 256) {
            return opcode == OP_PUSHDATA1;
        }
        if (data.length < 65536) {
            return opcode == OP_PUSHDATA2;
        }

        // can never be used, but implemented for completeness
        return opcode == OP_PUSHDATA4;
    }

    public void write(OutputStream stream) throws IOException {
        if (isOpCode()) {
            checkState(data == null);
            stream.write(opcode);
        } else if (data != null) {
            if (opcode < OP_PUSHDATA1) {
                checkState(data.length == opcode);
                stream.write(opcode);
            } else if (opcode == OP_PUSHDATA1) {
                checkState(data.length <= 0xFF);
                stream.write(OP_PUSHDATA1);
                stream.write(data.length);
            } else if (opcode == OP_PUSHDATA2) {
                checkState(data.length <= 0xFFFF);
                stream.write(OP_PUSHDATA2);
                stream.write(0xFF & data.length);
                stream.write(0xFF & (data.length >> 8));
            } else if (opcode == OP_PUSHDATA4) {
                checkState(data.length <= Script.MAX_SCRIPT_ELEMENT_SIZE);
                stream.write(OP_PUSHDATA4);
                SerializeUtils.uint32ToByteStreamLE(data.length, stream);
            } else {
                throw new RuntimeException("Unimplemented");
            }
```

```
            stream.write(data);
        } else {
            stream.write(opcode); // smallNum
        }
    }

    @Override
    public String toString() {
        StringBuilder buf = new StringBuilder();
        if (isOpCode()) {
            buf.append(getOpCodeName(opcode));
        } else if (data != null) {
            // Data chunk
buf.append(getPushDataName(opcode)).append("[").append(Hex.encode(data)).append("]");
} else {
            // Small num
            buf.append(Script.decodeFromOpN(opcode));
        }
        return buf.toString();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        ScriptChunk other = (ScriptChunk) o;
        return opcode == other.opcode && startLocationInProgram == other.startLocationInProgram
                && Arrays.equals(data, other.data);
    }

    @Override
    public int hashCode() {
        return Objects.hashCode(opcode, startLocationInProgram, Arrays.hashCode(data));
    }
}
```

62:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\script\ScriptException.java

```
 */

package io.nuls.kernel.script;

public class ScriptException extends RuntimeException {
    public ScriptException(String msg) {
        super(msg);
    }

    public ScriptException(String msg, Exception e) {
        super(msg, e);
    }
}

63:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\script\ScriptOpCodes.java
 * Various constants that define the assembly-like scripting language that forms part of the Bitcoin protocol.
 * See {@link org.bitcoinj.script.Script} for details. Also provides a method to convert them to a string.
 */
public class ScriptOpCodes {
    // push value
    public static final int OP_0 = 0x00; // push empty vector
    public static final int OP_FALSE = OP_0;
    public static final int OP_PUSHDATA1 = 0x4c;
    public static final int OP_PUSHDATA2 = 0x4d;
    public static final int OP_PUSHDATA4 = 0x4e;
    public static final int OP_1NEGATE = 0x4f;
    public static final int OP_RESERVED = 0x50;
    public static final int OP_1 = 0x51;
    public static final int OP_TRUE = OP_1;
    public static final int OP_2 = 0x52;
    public static final int OP_3 = 0x53;
    public static final int OP_4 = 0x54;
    public static final int OP_5 = 0x55;
    public static final int OP_6 = 0x56;
    public static final int OP_7 = 0x57;
    public static final int OP_8 = 0x58;
    public static final int OP_9 = 0x59;
    public static final int OP_10 = 0x5a;
    public static final int OP_11 = 0x5b;
```

```
public static final int OP_12 = 0x5c;
public static final int OP_13 = 0x5d;
public static final int OP_14 = 0x5e;
public static final int OP_15 = 0x5f;
public static final int OP_16 = 0x60;

// control
public static final int OP_NOP = 0x61;
public static final int OP_VER = 0x62;
public static final int OP_IF = 0x63;
public static final int OP_NOTIF = 0x64;
public static final int OP_VERIF = 0x65;
public static final int OP_VERNOTIF = 0x66;
public static final int OP_ELSE = 0x67;
public static final int OP_ENDIF = 0x68;
public static final int OP_VERIFY = 0x69;
public static final int OP_RETURN = 0x6a;

// stack ops
public static final int OP_TOALTSTACK = 0x6b;
public static final int OP_FROMALTSTACK = 0x6c;
public static final int OP_2DROP = 0x6d;
public static final int OP_2DUP = 0x6e;
public static final int OP_3DUP = 0x6f;
public static final int OP_2OVER = 0x70;
public static final int OP_2ROT = 0x71;
public static final int OP_2SWAP = 0x72;
public static final int OP_IFDUP = 0x73;
public static final int OP_DEPTH = 0x74;
public static final int OP_DROP = 0x75;
public static final int OP_DUP = 0x76;
public static final int OP_NIP = 0x77;
public static final int OP_OVER = 0x78;
public static final int OP_PICK = 0x79;
public static final int OP_ROLL = 0x7a;
public static final int OP_ROT = 0x7b;
public static final int OP_SWAP = 0x7c;
public static final int OP_TUCK = 0x7d;

// splice ops
public static final int OP_CAT = 0x7e;
public static final int OP_SUBSTR = 0x7f;
```

```
public static final int OP_LEFT = 0x80;
public static final int OP_RIGHT = 0x81;
public static final int OP_SIZE = 0x82;

// bit logic
public static final int OP_INVERT = 0x83;
public static final int OP_AND = 0x84;
public static final int OP_OR = 0x85;
public static final int OP_XOR = 0x86;
public static final int OP_EQUAL = 0x87;
public static final int OP_EQUALVERIFY = 0x88;
public static final int OP_RESERVED1 = 0x89;
public static final int OP_RESERVED2 = 0x8a;

// numeric
public static final int OP_1ADD = 0x8b;
public static final int OP_1SUB = 0x8c;
public static final int OP_2MUL = 0x8d;
public static final int OP_2DIV = 0x8e;
public static final int OP_NEGATE = 0x8f;
public static final int OP_ABS = 0x90;
public static final int OP_NOT = 0x91;
public static final int OP_0NOTEQUAL = 0x92;
public static final int OP_ADD = 0x93;
public static final int OP_SUB = 0x94;
public static final int OP_MUL = 0x95;
public static final int OP_DIV = 0x96;
public static final int OP_MOD = 0x97;
public static final int OP_LSHIFT = 0x98;
public static final int OP_RSHIFT = 0x99;
public static final int OP_BOOLAND = 0x9a;
public static final int OP_BOOLOR = 0x9b;
public static final int OP_NUMEQUAL = 0x9c;
public static final int OP_NUMEQUALVERIFY = 0x9d;
public static final int OP_NUMNOTEQUAL = 0x9e;
public static final int OP_LESSTHAN = 0x9f;
public static final int OP_GREATERTHAN = 0xa0;
public static final int OP_LESSTHANOREQUAL = 0xa1;
public static final int OP_GREATERTHANOREQUAL = 0xa2;
public static final int OP_MIN = 0xa3;
public static final int OP_MAX = 0xa4;
public static final int OP_WITHIN = 0xa5;
```

```
// crypto
public static final int OP_RIPEMD160 = 0xa6;
public static final int OP_SHA1 = 0xa7;
public static final int OP_SHA256 = 0xa8;
public static final int OP_HASH160 = 0xa9;
public static final int OP_HASH256 = 0xaa;
public static final int OP_CODESEPARATOR = 0xab;
public static final int OP_CHECKSIG = 0xac;
public static final int OP_CHECKSIGVERIFY = 0xad;
public static final int OP_CHECKMULTISIG = 0xae;
public static final int OP_CHECKMULTISIGVERIFY = 0xaf;

// block state
/**
 * Check lock time of the block. Introduced in BIP 65, replacing OP_NOP2
 */
public static final int OP_CHECKLOCKTIMEVERIFY = 0xb1;

// expansion
public static final int OP_NOP1 = 0xb0;
/**
 * Deprecated by BIP 65
 */
@Deprecated
public static final int OP_NOP2 = OP_CHECKLOCKTIMEVERIFY;
public static final int OP_NOP3 = 0xb2;
public static final int OP_NOP4 = 0xb3;
public static final int OP_NOP5 = 0xb4;
public static final int OP_NOP6 = 0xb5;
public static final int OP_NOP7 = 0xb6;
public static final int OP_NOP8 = 0xb7;
public static final int OP_NOP9 = 0xb8;
public static final int OP_NOP10 = 0xb9;
public static final int OP_INVALIDOPCODE = 0xff;

private static final Map<Integer, String> opCodeMap = ImmutableMap.<Integer, String>builder()
        .put(OP_0, "0")
        .put(OP_PUSHDATA1, "PUSHDATA1")
        .put(OP_PUSHDATA2, "PUSHDATA2")
        .put(OP_PUSHDATA4, "PUSHDATA4")
        .put(OP_1NEGATE, "1NEGATE")
```

```
.put(OP_RESERVED, "RESERVED")
.put(OP_1, "1")
.put(OP_2, "2")
.put(OP_3, "3")
.put(OP_4, "4")
.put(OP_5, "5")
.put(OP_6, "6")
.put(OP_7, "7")
.put(OP_8, "8")
.put(OP_9, "9")
.put(OP_10, "10")
.put(OP_11, "11")
.put(OP_12, "12")
.put(OP_13, "13")
.put(OP_14, "14")
.put(OP_15, "15")
.put(OP_16, "16")
.put(OP_NOP, "NOP")
.put(OP_VER, "VER")
.put(OP_IF, "IF")
.put(OP_NOTIF, "NOTIF")
.put(OP_VERIF, "VERIF")
.put(OP_VERNOTIF, "VERNOTIF")
.put(OP_ELSE, "ELSE")
.put(OP_ENDIF, "ENDIF")
.put(OP_VERIFY, "VERIFY")
.put(OP_RETURN, "RETURN")
.put(OP_TOALTSTACK, "TOALTSTACK")
.put(OP_FROMALTSTACK, "FROMALTSTACK")
.put(OP_2DROP, "2DROP")
.put(OP_2DUP, "2DUP")
.put(OP_3DUP, "3DUP")
.put(OP_2OVER, "2OVER")
.put(OP_2ROT, "2ROT")
.put(OP_2SWAP, "2SWAP")
.put(OP_IFDUP, "IFDUP")
.put(OP_DEPTH, "DEPTH")
.put(OP_DROP, "DROP")
.put(OP_DUP, "DUP")
.put(OP_NIP, "NIP")
.put(OP_OVER, "OVER")
.put(OP_PICK, "PICK")
```

```
.put(OP_ROLL, "ROLL")
.put(OP_ROT, "ROT")
.put(OP_SWAP, "SWAP")
.put(OP_TUCK, "TUCK")
.put(OP_CAT, "CAT")
.put(OP_SUBSTR, "SUBSTR")
.put(OP_LEFT, "LEFT")
.put(OP_RIGHT, "RIGHT")
.put(OP_SIZE, "SIZE")
.put(OP_INVERT, "INVERT")
.put(OP_AND, "AND")
.put(OP_OR, "OR")
.put(OP_XOR, "XOR")
.put(OP_EQUAL, "EQUAL")
.put(OP_EQUALVERIFY, "EQUALVERIFY")
.put(OP_RESERVED1, "RESERVED1")
.put(OP_RESERVED2, "RESERVED2")
.put(OP_1ADD, "1ADD")
.put(OP_1SUB, "1SUB")
.put(OP_2MUL, "2MUL")
.put(OP_2DIV, "2DIV")
.put(OP_NEGATE, "NEGATE")
.put(OP_ABS, "ABS")
.put(OP_NOT, "NOT")
.put(OP_0NOTEQUAL, "0NOTEQUAL")
.put(OP_ADD, "ADD")
.put(OP_SUB, "SUB")
.put(OP_MUL, "MUL")
.put(OP_DIV, "DIV")
.put(OP_MOD, "MOD")
.put(OP_LSHIFT, "LSHIFT")
.put(OP_RSHIFT, "RSHIFT")
.put(OP_BOOLAND, "BOOLAND")
.put(OP_BOOLOR, "BOOLOR")
.put(OP_NUMEQUAL, "NUMEQUAL")
.put(OP_NUMEQUALVERIFY, "NUMEQUALVERIFY")
.put(OP_NUMNOTEQUAL, "NUMNOTEQUAL")
.put(OP_LESSTHAN, "LESSTHAN")
.put(OP_GREATERTHAN, "GREATERTHAN")
.put(OP_LESSTHANOREQUAL, "LESSTHANOREQUAL")
.put(OP_GREATERTHANOREQUAL, "GREATERTHANOREQUAL")
.put(OP_MIN, "MIN")
```

```
            .put(OP_MAX, "MAX")
            .put(OP_WITHIN, "WITHIN")
            .put(OP_RIPEMD160, "RIPEMD160")
            .put(OP_SHA1, "SHA1")
            .put(OP_SHA256, "SHA256")
            .put(OP_HASH160, "HASH160")
            .put(OP_HASH256, "HASH256")
            .put(OP_CODESEPARATOR, "CODESEPARATOR")
            .put(OP_CHECKSIG, "CHECKSIG")
            .put(OP_CHECKSIGVERIFY, "CHECKSIGVERIFY")
            .put(OP_CHECKMULTISIG, "CHECKMULTISIG")
            .put(OP_CHECKMULTISIGVERIFY, "CHECKMULTISIGVERIFY")
            .put(OP_NOP1, "NOP1")
            .put(OP_CHECKLOCKTIMEVERIFY, "CHECKLOCKTIMEVERIFY")
            .put(OP_NOP3, "NOP3")
            .put(OP_NOP4, "NOP4")
            .put(OP_NOP5, "NOP5")
            .put(OP_NOP6, "NOP6")
            .put(OP_NOP7, "NOP7")
            .put(OP_NOP8, "NOP8")
            .put(OP_NOP9, "NOP9")
            .put(OP_NOP10, "NOP10").build();

    private static final Map<String, Integer> opCodeNameMap = ImmutableMap.<String,
Integer>builder()
            .put("0", OP_0)
            .put("PUSHDATA1", OP_PUSHDATA1)
            .put("PUSHDATA2", OP_PUSHDATA2)
            .put("PUSHDATA4", OP_PUSHDATA4)
            .put("1NEGATE", OP_1NEGATE)
            .put("RESERVED", OP_RESERVED)
            .put("1", OP_1)
            .put("2", OP_2)
            .put("3", OP_3)
            .put("4", OP_4)
            .put("5", OP_5)
            .put("6", OP_6)
            .put("7", OP_7)
            .put("8", OP_8)
            .put("9", OP_9)
            .put("10", OP_10)
            .put("11", OP_11)
```

```
.put("12", OP_12)
.put("13", OP_13)
.put("14", OP_14)
.put("15", OP_15)
.put("16", OP_16)
.put("NOP", OP_NOP)
.put("VER", OP_VER)
.put("IF", OP_IF)
.put("NOTIF", OP_NOTIF)
.put("VERIF", OP_VERIF)
.put("VERNOTIF", OP_VERNOTIF)
.put("ELSE", OP_ELSE)
.put("ENDIF", OP_ENDIF)
.put("VERIFY", OP_VERIFY)
.put("RETURN", OP_RETURN)
.put("TOALTSTACK", OP_TOALTSTACK)
.put("FROMALTSTACK", OP_FROMALTSTACK)
.put("2DROP", OP_2DROP)
.put("2DUP", OP_2DUP)
.put("3DUP", OP_3DUP)
.put("2OVER", OP_2OVER)
.put("2ROT", OP_2ROT)
.put("2SWAP", OP_2SWAP)
.put("IFDUP", OP_IFDUP)
.put("DEPTH", OP_DEPTH)
.put("DROP", OP_DROP)
.put("DUP", OP_DUP)
.put("NIP", OP_NIP)
.put("OVER", OP_OVER)
.put("PICK", OP_PICK)
.put("ROLL", OP_ROLL)
.put("ROT", OP_ROT)
.put("SWAP", OP_SWAP)
.put("TUCK", OP_TUCK)
.put("CAT", OP_CAT)
.put("SUBSTR", OP_SUBSTR)
.put("LEFT", OP_LEFT)
.put("RIGHT", OP_RIGHT)
.put("SIZE", OP_SIZE)
.put("INVERT", OP_INVERT)
.put("AND", OP_AND)
.put("OR", OP_OR)
```

```
.put("XOR", OP_XOR)
.put("EQUAL", OP_EQUAL)
.put("EQUALVERIFY", OP_EQUALVERIFY)
.put("RESERVED1", OP_RESERVED1)
.put("RESERVED2", OP_RESERVED2)
.put("1ADD", OP_1ADD)
.put("1SUB", OP_1SUB)
.put("2MUL", OP_2MUL)
.put("2DIV", OP_2DIV)
.put("NEGATE", OP_NEGATE)
.put("ABS", OP_ABS)
.put("NOT", OP_NOT)
.put("0NOTEQUAL", OP_0NOTEQUAL)
.put("ADD", OP_ADD)
.put("SUB", OP_SUB)
.put("MUL", OP_MUL)
.put("DIV", OP_DIV)
.put("MOD", OP_MOD)
.put("LSHIFT", OP_LSHIFT)
.put("RSHIFT", OP_RSHIFT)
.put("BOOLAND", OP_BOOLAND)
.put("BOOLOR", OP_BOOLOR)
.put("NUMEQUAL", OP_NUMEQUAL)
.put("NUMEQUALVERIFY", OP_NUMEQUALVERIFY)
.put("NUMNOTEQUAL", OP_NUMNOTEQUAL)
.put("LESSTHAN", OP_LESSTHAN)
.put("GREATERTHAN", OP_GREATERTHAN)
.put("LESSTHANOREQUAL", OP_LESSTHANOREQUAL)
.put("GREATERTHANOREQUAL", OP_GREATERTHANOREQUAL)
.put("MIN", OP_MIN)
.put("MAX", OP_MAX)
.put("WITHIN", OP_WITHIN)
.put("RIPEMD160", OP_RIPEMD160)
.put("SHA1", OP_SHA1)
.put("SHA256", OP_SHA256)
.put("HASH160", OP_HASH160)
.put("HASH256", OP_HASH256)
.put("CODESEPARATOR", OP_CODESEPARATOR)
.put("CHECKSIG", OP_CHECKSIG)
.put("CHECKSIGVERIFY", OP_CHECKSIGVERIFY)
.put("CHECKMULTISIG", OP_CHECKMULTISIG)
.put("CHECKMULTISIGVERIFY", OP_CHECKMULTISIGVERIFY)
```

```
        .put("NOP1", OP_NOP1)
        .put("CHECKLOCKTIMEVERIFY", OP_CHECKLOCKTIMEVERIFY)
        .put("NOP2", OP_NOP2)
        .put("NOP3", OP_NOP3)
        .put("NOP4", OP_NOP4)
        .put("NOP5", OP_NOP5)
        .put("NOP6", OP_NOP6)
        .put("NOP7", OP_NOP7)
        .put("NOP8", OP_NOP8)
        .put("NOP9", OP_NOP9)
        .put("NOP10", OP_NOP10).build();

/**
 * Converts the given OpCode into a string (eg "0", "PUSHDATA", or "NON_OP(10)")
 */
public static String getOpCodeName(int opcode) {
    if (opCodeMap.containsKey(opcode)) {
        return opCodeMap.get(opcode);
    }

    return "NON_OP(" + opcode + ")";
}

/**
 * Converts the given pushdata OpCode into a string (eg "PUSHDATA2", or "PUSHDATA(23)")
 */
public static String getPushDataName(int opcode) {
    if (opCodeMap.containsKey(opcode)) {
        return opCodeMap.get(opcode);
    }

    return "PUSHDATA(" + opcode + ")";
}

/**
 * Converts the given OpCodeName into an int
 */
public static int getOpCode(String opCodeName) {
    if (opCodeNameMap.containsKey(opCodeName)) {
        return opCodeNameMap.get(opCodeName);
    }
```

```
        return OP_INVALIDOPCODE;
    }
}

64:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\ScriptSign.java
 */
package io.nuls.kernel.script;

import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BaseNulsData;
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.ArrayList;
import java.util.List;

public class ScriptSign extends BaseNulsData {

    private List<Script> scripts;

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        if (scripts != null && scripts.size() > 0) {
            for (Script script : scripts) {
                stream.writeBytesWithLength(script.getProgram());
            }
        } else {
            stream.write(NulsConstant.PLACE_HOLDER);
        }
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        List<Script> scripts = new ArrayList<>();
        while (!byteBuffer.isFinished()) {
            scripts.add(new Script(byteBuffer.readByLengthByte()));
        }
        this.scripts = scripts;
```

```
    }

    public static ScriptSign createFromBytes(byte[] bytes) throws NulsException {
        ScriptSign sig = new ScriptSign();
        sig.parse(bytes, 0);
        return sig;
    }

    public List<Script> getScripts() {
        return scripts;
    }

    public void setScripts(List<Script> scripts) {
        this.scripts = scripts;
    }

    @Override
    public int size() {
        int size = 0;
        if (scripts != null && scripts.size() > 0) {
            for (Script script : scripts) {
                size += SerializeUtils.sizeOfBytes(script.getProgram());
            }
        } else {
            size = 4;
        }
        return size;
    }

}

65:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\ScriptUtil.java
 */
package io.nuls.kernel.script;

import io.nuls.kernel.utils.SerializeUtils;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
public class ScriptUtil {

    /**
     *  P2PSH
     *
     * @param sigByte
     * @param pubkeyByte
     * @return Script
     */
    public static Script createP2PKHInputScript(byte[] sigByte, byte[] pubkeyByte) {
        return ScriptBuilder.createNulsInputScript(sigByte, pubkeyByte);
    }

    /**
     *
     *
     * @param address
     * @return Script
     */
    public static Script createP2PKHOutputScript(byte[] address) {
        return ScriptBuilder.createOutputScript(address, 1);
    }


    /**
     * M-NM-N
     *
     * @param pub_keys
     * @param m
     * @return Script
     */
    public static Script creatRredeemScript(List<String> pub_keys, int m) {
        return ScriptBuilder.createNulsRedeemScript(m, pub_keys);
    }

    /**
     * M-NM-NN
     *
     * @param signatures
     * @param multisigProgram P2SH
     * @return Script
     */
```

```
public static Script createP2SHInputScript(List<byte[]> signatures, Script multisigProgram) {
    return ScriptBuilder.createNulsP2SHMultiSigInputScript(signatures, multisigProgram);
}

/**
 * M-NM-NN
 *
 * @param redeemScript
 * @return Script
 */
public static Script createP2SHOutputScript(Script redeemScript) {
    return ScriptBuilder.createP2SHOutputScript(redeemScript);
}

/**
 * M-N
 *
 * @param address
 * @return Script
 */
public static Script createP2SHOutputScript(byte[] address) {
    return ScriptBuilder.createOutputScript(address, 0);
}

public static void main(String[] args) {
    /**
     *
     * */
    try {
        /*TransferTransaction tx = new TransferTransaction();
        tx.setTime(TimeService.currentTimeMillis());

        CoinData coinData = new CoinData();
        List<Coin> from = new ArrayList<Coin>();
        for(int i=0;i<3;i++){
            String addr = "tx_hash+index"+1;
            Coin from_coin = new Coin(addr.getBytes(),Na.valueOf(100));
            from.add(from_coin);
        }
        List<Coin> to = new ArrayList<Coin>();
        for(int i=0;i<3;i++){
            String addr = "Nsdybg1xmP7z4PTUKKN26stocrJ1qrU"+1;
```

```
        Coin to_coin = new Coin(AddressTool.getAddress(addr),Na.valueOf(100));
        to_coin.setScript(ScriptBuilder.createOutputScript(AddressTool.getAddress(addr),0));
        to.add(to_coin);
    }
    coinData.setFrom(from);
    coinData.setTo(to);
    tx.setCoinData(coinData);
    tx.setHash(NulsDigestData.calcDigestData(tx.serializeForHash()));
    P2PKHScriptSig sig = new P2PKHScriptSig();
    List<Script> scripts = new ArrayList<Script>();
    for(int i=0;i<3;i++){
        String addr = "Nse84JjkxBoR9zXLGftu8X8xjsfnwxW"+1;
        scripts.add(ScriptBuilder.createOutputScript(AddressTool.getAddress(addr),1));
    }
    sig.setScripts(scripts);
    sig.setPublicKey("publickey".getBytes());

    sig.setSignData(new NulsSignData());
    tx.setBlockSignature(sig.serialize());

    byte[] bytes = tx.serialize();


    TransferTransaction tx2 = new TransferTransaction();
    tx2.parse(new NulsByteBuffer(bytes));
    System.out.println(bytes.length);
    for (Coin coin : tx2.getCoinData().getTo()) {
        System.out.println(coin.getNa());
        System.out.println(coin.getScript().getChunks());
    }
    P2PKHScriptSig scriptSig = new P2PKHScriptSig();
    scriptSig.parse(new NulsByteBuffer(tx2.getBlockSignature()));
    for (Script script:scriptSig.getScripts()) {
        System.out.println(script.getChunks());
    }*/

    /**
     *
     * */
    //P2PKHInput
    byte[] signbyte =
"cVLwRLTvz3BxDAWkvS3yzT9pUcTCup7kQnfT2smRjvmmm1wAP6QT".getBytes();
```

```
        byte[] pubkeyByte = "public_key".getBytes();
        Script inputScript = createP2PKHInputScript(signbyte, pubkeyByte);
        System.out.println("P2PKH_INPUT:" + inputScript.getChunks());
        //System.out.println(new String(inputScript.getChunks().get(0).data));
        //P2PKHOutput
        byte[] addrByte = "Nsdybg1xmP7z4PTUKKN26stocrJ1qrUJ".getBytes();
        Script outputScript = createP2PKHOutputScript(addrByte);
        System.out.println("P2PKH_OUTPUT:" + outputScript.getChunks());
        //redeemScript
        List<String> pub_keys = new ArrayList<String>();
        for (int i = 0; i < 3; i++) {
            pub_keys.add("Nsdybg1xmP7z4PTUKKN26stocrJ1qrU" + i);
        }
        Script redeemScript = creatRredeemScript(pub_keys, 2);
        System.out.println("REDEEM:" + redeemScript.getChunks());
        //P2SHInput
        List<byte[]> signBytes = new ArrayList<byte[]>();
        for (int i = 0; i < 3; i++) {
signBytes.add("cVLwRLTvz3BxDAWkvS3yzT9pUcTCup7kQnfT2smRjvmmm1wAP6Q".getBytes())
;
        }
        System.out.println(redeemScript.getProgram().length);
        Script p2shInput = createP2SHInputScript(signBytes, redeemScript);
        System.out.println("P2SH_INPUT:" + p2shInput.getChunks());
        ScriptChunk scriptChunk = p2shInput.getChunks().get(p2shInput.getChunks().size() - 1);
//scriptChunk.data
        Script redeemScriptParse = new Script(scriptChunk.data);
        System.out.println(redeemScriptParse.getChunks());
        //P2SHOutput
        Script p2shOutput = createP2SHOutputScript(redeemScript);
        System.out.println("P2SH_OUTPUT:" + p2shOutput.getChunks());

System.out.println(Arrays.toString(SerializeUtils.sha256hash160("03a690c7f3b07e320566162b0ff
7d79c8c9f453c0a4a13305fcd90f4e4f4cf215c".getBytes())));

        /**
         * P2PKH
         * */
        /* Na values = Na.valueOf(10);
        byte[] from = "".getBytes();                    //
        byte[] to   = "".getBytes();                    //
        String pub_key = "";                            //
```

```
            String password ="";
            String remark ="";
            Na price = Na.valueOf(5);
            TransferTransaction tx = new TransferTransaction();
            tx.setTime(TimeService.currentTimeMillis());
            CoinData coinData = new CoinData();
            Coin toCoin = new Coin(to, values);
            coinData.getTo().add(toCoin);
            if (price == null) {
                price = TransactionFeeCalculator.MIN_PRECE_PRE_1024_BYTES;
            }
            CoinDataResult coinDataResult = accountLedgerService.getCoinData(from, values,
tx.size() +  + coinData.size(), price);
            if (!coinDataResult.isEnough()) {
                //return Result.getFailed(AccountLedgerErrorCode.INSUFFICIENT_BALANCE);
                System.out.println("");
                return;
            }
            coinData.setFrom(coinDataResult.getCoinList());
            if (coinDataResult.getChange() != null) {
                coinData.getTo().add(coinDataResult.getChange());
            }
            tx.setCoinData(coinData);
            tx.setHash(NulsDigestData.calcDigestData(tx.serializeForHash()));
            P2PKHScriptSig sig = new P2PKHScriptSig();
            //sig.setPublicKey(account.getPubKey());*/
            //hash
            //sig.setSignData(accountService.signDigest(tx.getHash().getDigestBytes(), account,
password));
            //tx.setBlockSignature(sig.serialize());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

66:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\SignatureUtil.java
 */
package io.nuls.kernel.script;
```

```
import io.nuls.core.tools.crypto.ECKey;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.lite.annotation.Component;
import io.nuls.kernel.model.*;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.SerializeUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.util.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

@Component
public class SignatureUtil {
    private static final Logger log = LoggerFactory.getLogger(SignatureUtil.class);

    /**
     *
     *
     * @param tx
     */
    public static boolean validateTransactionSignture(Transaction tx) throws NulsException {
        try {
            if (!tx.needVerifySignature()) {
                return true;
            }
            if (tx.getTransactionSignature() == null && tx.getTransactionSignature().length == 0) {
                throw new NulsException(KernelErrorCode.SIGNATURE_ERROR);
            }
            TransactionSignature transactionSignature = new TransactionSignature();
            transactionSignature.parse(tx.getTransactionSignature(), 0);
            if ((transactionSignature.getP2PHKSignatures() == null ||
transactionSignature.getP2PHKSignatures().size() == 0)
                    && (transactionSignature.getScripts() == null ||
transactionSignature.getScripts().size() == 0)) {
                throw new NulsException(KernelErrorCode.SIGNATURE_ERROR);
            }
            if (transactionSignature.getP2PHKSignatures() != null &&
```

```
transactionSignature.getP2PHKSignatures().size() > 0) {
            for (P2PHKSignature signature : transactionSignature.getP2PHKSignatures()) {
                if (!ECKey.verify(tx.getHash().getDigestBytes(),
signature.getSignData().getSignBytes(), signature.getPublicKey())) {
                    throw new NulsException(KernelErrorCode.SIGNATURE_ERROR);
                }
            }
        }
        if (transactionSignature.getScripts() != null && transactionSignature.getScripts().size() > 0)
{
            for (Script script : transactionSignature.getScripts()) {
                if (!validScriptSign(tx.getHash().getDigestBytes(), script.getChunks())) {
                    throw new NulsException(KernelErrorCode.SIGNATURE_ERROR);
                }
            }
        }
    } catch (NulsException e) {
        log.error("TransactionSignature parse error!");
        throw e;
    }
    return true;
}

/**
 *
 *
 * @param tx
 */
public static boolean containsAddress(Transaction tx, byte[] address) throws NulsException {
    Set<String> addressSet = getAddressFromTX(tx);
    if (addressSet == null || addressSet.size() == 0) {
        return false;
    }
    if (addressSet.contains(AddressTool.getStringAddressByBytes(address))) {
        return true;
    }
    return false;
}

/**
 *
 *
```

```
     * @param tx
     */
    public static Set<String> getAddressFromTX(Transaction tx) throws NulsException {
        Set<String> addressSet = new HashSet<>();
        if (tx.getTransactionSignature() == null && tx.getTransactionSignature().length == 0) {
            return null;
        }
        try {
            TransactionSignature transactionSignature = new TransactionSignature();
            transactionSignature.parse(tx.getTransactionSignature(), 0);
            if ((transactionSignature.getP2PHKSignatures() == null ||
transactionSignature.getP2PHKSignatures().size() == 0) && (transactionSignature.getScripts() ==
null || transactionSignature.getScripts().size() == 0)) {
                return null;
            }
            if (transactionSignature.getP2PHKSignatures() != null &&
transactionSignature.getP2PHKSignatures().size() > 0) {
                for (P2PHKSignature signature : transactionSignature.getP2PHKSignatures()) {
                    if (signature.getPublicKey() != null || signature.getPublicKey().length == 0) {
addressSet.add(AddressTool.getStringAddressByBytes(AddressTool.getAddress(signature.getPu
blicKey())));
                    }
                }
            }
            if (transactionSignature.getScripts() != null && transactionSignature.getScripts().size() > 0)
{
                for (Script script : transactionSignature.getScripts()) {
                    if (script != null && script.getChunks() != null && script.getChunks().size() >= 2) {
                        addressSet.add(getScriptAddress(script.getChunks()));
                    }
                }
            }
        } catch (NulsException e) {
            log.error("TransactionSignature parse error!");
            throw e;
        }
        return addressSet;
    }

    /**
     * TransactionSignture
     *
```

```
     * @param tx
     * @param scriptEckeys
     * @param signEckeys
     */
    public static void createTransactionSignture(Transaction tx, List<ECKey> scriptEckeys,
List<ECKey> signEckeys) throws IOException {
        TransactionSignature transactionSignature = new TransactionSignature();
        List<P2PHKSignature> p2PHKSignatures = null;
        List<Script> scripts = null;
        try {
            if (scriptEckeys != null && scriptEckeys.size() > 0) {
                List<byte[]> signtures = new ArrayList<>();
                List<byte[]> pubkeys = new ArrayList<>();
                for (ECKey ecKey : scriptEckeys) {
                    signtures.add(signDigest(tx.getHash().getDigestBytes(), ecKey).getSignBytes());
                    pubkeys.add(ecKey.getPubKey());
                }
                scripts = createInputScripts(signtures, pubkeys);
            }
            if (signEckeys != null && signEckeys.size() > 0) {
                p2PHKSignatures = createSignaturesByEckey(tx, signEckeys);
            }
            transactionSignature.setP2PHKSignatures(p2PHKSignatures);
            transactionSignature.setScripts(scripts);
            tx.setTransactionSignature(transactionSignature.serialize());
        } catch (IOException ie) {
            log.error("TransactionSignature serialize error!");
            throw ie;
        }
    }

    /**
     *
     *
     * @param tx
     * @param eckeys
     */
    public static List<P2PHKSignature> createSignaturesByEckey(Transaction tx, List<ECKey>
eckeys) {
        List<P2PHKSignature> signatures = new ArrayList<>();
        for (ECKey ecKey : eckeys) {
            signatures.add(createSignatureByEckey(tx, ecKey));
```

```
    }
    return signatures;
}

/**
 *
 *
 * @param tx
 * @param ecKey
 */
public static P2PHKSignature createSignatureByEckey(Transaction tx, ECKey ecKey) {
    P2PHKSignature p2PHKSignature = new P2PHKSignature();
    p2PHKSignature.setPublicKey(ecKey.getPubKey());
    //hash
    p2PHKSignature.setSignData(signDigest(tx.getHash().getDigestBytes(), ecKey));
    return p2PHKSignature;
}

/**
 *
 *
 * @param signtures
 * @param pubkeys
 */
public static List<Script> createInputScripts(List<byte[]> signtures, List<byte[]> pubkeys) {
    List<Script> scripts = new ArrayList<>();
    if (signtures == null || pubkeys == null || signtures.size() != pubkeys.size()) {
        return null;
    }
    //
    for (int i = 0; i < signtures.size(); i++) {
        scripts.add(createInputScript(signtures.get(i), pubkeys.get(i)));
    }
    return scripts;
}

/**
 *
 *
 * @param signture
 * @param pubkey
 */
```

```
public static Script createInputScript(byte[] signture, byte[] pubkey) {
    Script script = null;
    if (signture != null && pubkey != null) {
        script = ScriptBuilder.createNulsInputScript(signture, pubkey);
    }
    return script;
}

/**
 *
 */
public static Script createOutputScript(byte[] address) {
    Script script = null;
    if (address == null || address.length < 23) {
        return null;
    }
    //
    if (address[2] == NulsContext.P2SH_ADDRESS_TYPE) {
        script = ScriptBuilder.createOutputScript(address, 0);
    } else {
        script = ScriptBuilder.createOutputScript(address, 1);
    }
    return script;
}

/**
 *
 *
 * @param tx
 */
public static boolean createOutputScript(Transaction tx) {
    CoinData coinData = tx.getCoinData();
    //
    for (Coin coin : coinData.getTo()) {
        Script scriptPubkey = null;
        byte[] toAddr = coin.getAddress();
        if (toAddr[2] == NulsContext.DEFAULT_ADDRESS_TYPE) {
            scriptPubkey = ScriptUtil.createP2PKHOutputScript(toAddr);
        } else if (toAddr[2] == NulsContext.P2SH_ADDRESS_TYPE) {
            scriptPubkey = ScriptUtil.createP2SHOutputScript(toAddr);
        }
        if (scriptPubkey != null && scriptPubkey.getProgram().length > 0) {
```

```
            coin.setOwner(scriptPubkey.getProgram());
        }
    }
    return true;
}

/**
 * P2SH
 *
 * @param signtures
 * @param pubkeys
 */
public static Script createP2shScript(List<byte[]> signtures, List<byte[]> pubkeys, int m) {
    Script scriptSig = null;
    //
    Script redeemScript = ScriptBuilder.createByteNulsRedeemScript(m, pubkeys);
    //
    scriptSig = ScriptBuilder.createNulsP2SHMultiSigInputScript(signtures, redeemScript);
    return scriptSig;
}


/**
 * P2SH
 *
 * @param digestBytes
 * @param chunks
 */
public static boolean validScriptSign(byte[] digestBytes, List<ScriptChunk> chunks) {
    if (chunks == null || chunks.size() < 2) {
        return false;
    }
    //OP_0/P2SH
    if (chunks.get(0).opcode == ScriptOpCodes.OP_0) {
        byte[] redeemByte = chunks.get(chunks.size() - 1).data;
        Script redeemScript = new Script(redeemByte);
        List<ScriptChunk> redeemChunks = redeemScript.getChunks();

        LinkedList<byte[]> signtures = new LinkedList<byte[]>();
        for (int i = 1; i < chunks.size() - 1; i++) {
            signtures.add(chunks.get(i).data);
        }
```

```
        LinkedList<byte[]> pubkeys = new LinkedList<byte[]>();
        int m = Script.decodeFromOpN(redeemChunks.get(0).opcode);
        if (signtures.size() < m) {
            return false;
        }

        for (int j = 1; j < redeemChunks.size() - 2; j++) {
            pubkeys.add(redeemChunks.get(j).data);
        }

        int n = Script.decodeFromOpN(redeemChunks.get(redeemChunks.size() - 2).opcode);
        if (n != pubkeys.size() || n < m) {
            return false;
        }
        return validMultiScriptSign(digestBytes, signtures, pubkeys);
    } else {
        if (!ECKey.verify(digestBytes, chunks.get(0).data, chunks.get(1).data)) {
            return false;
        }
    }
    return true;
}


/**
 *
 *
 * @param redeemScript
 */
public static int getM(Script redeemScript) {
    return Script.decodeFromOpN(redeemScript.getChunks().get(0).opcode);
}

/**
 *
 */
public static String getScriptAddress(List<ScriptChunk> chunks) {
    if (chunks.get(0).opcode == ScriptOpCodes.OP_0) {
        byte[] redeemByte = chunks.get(chunks.size() - 1).data;
        Script redeemScript = new Script(redeemByte);
        Address address = new Address(NulsContext.DEFAULT_CHAIN_ID,
```

```
NulsContext.P2SH_ADDRESS_TYPE,
SerializeUtils.sha256hash160(redeemScript.getProgram()));
            return address.toString();
        } else {
            return
AddressTool.getStringAddressByBytes(AddressTool.getAddress(chunks.get(1).data));
        }
    }

    /**
     *
     *
     * @param digestBytes
     * @param signtures
     */
    public static boolean validMultiScriptSign(byte[] digestBytes, LinkedList<byte[]> signtures,
LinkedList<byte[]> pubkeys) {
        while (signtures.size() > 0) {
            byte[] pubKey = pubkeys.pollFirst();
            if (ECKey.verify(digestBytes, signtures.getFirst(), pubKey)) {
                signtures.pollFirst();
            }
            if (signtures.size() > pubkeys.size()) {
                return false;
            }
        }
        return true;
    }

    /**
     *
     *
     * @param digest
     * @param ecKey
     */
    public static NulsSignData signDigest(byte[] digest, ECKey ecKey) {
        byte[] signbytes = ecKey.sign(digest);
        NulsSignData nulsSignData = new NulsSignData();
        nulsSignData.setSignAlgType(NulsSignData.SIGN_ALG_ECC);
        nulsSignData.setSignBytes(signbytes);
        return nulsSignData;
    }
```

```
}

67:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\script\TransactionSignature.java
 */

package io.nuls.kernel.script;

import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BaseNulsData;
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.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TransactionSignature extends BaseNulsData {
    private List<P2PHKSignature> p2PHKSignatures;
    private List<Script> scripts;

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        //
        if (p2PHKSignatures != null && p2PHKSignatures.size() > 0) {
            for (P2PHKSignature p2PHKSignature : p2PHKSignatures) {
                if (p2PHKSignature != null) {
                    stream.writeNulsData(p2PHKSignature);
                }
            }
        }
        if (scripts != null && scripts.size() > 0) {
            //
            stream.write(NulsConstant.SIGN_HOLDER);
            //
            for (Script script : scripts) {
                if (script != null && script.getProgram() != null && script.getProgram().length > 0) {
                    stream.writeBytesWithLength(script.getProgram());
```

```
            }
        }
    }
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    // ,
    int course = 0;
    boolean isScript = false;
    List<P2PHKSignature> p2PHKSignatures = new ArrayList<>();
    List<Script> scripts = new ArrayList<>();
    while (!byteBuffer.isFinished()) {
        course = byteBuffer.getCursor();
        //0x00
        if (!isScript && byteBuffer.getPayload().length < 2) {
            break;
        }
        if (isScript || Arrays.equals(NulsConstant.SIGN_HOLDER, byteBuffer.readBytes(2))) {
            isScript = true;
            if (!byteBuffer.isFinished()) {
                scripts.add(new Script(byteBuffer.readByLengthByte()));
            }
        } else {
            byteBuffer.setCursor(course);
            p2PHKSignatures.add(byteBuffer.readNulsData(new P2PHKSignature()));
        }
    }
    this.p2PHKSignatures = p2PHKSignatures;
    this.scripts = scripts;
}

@Override
public int size() {
    //
    int size = 0;
    if (p2PHKSignatures != null && p2PHKSignatures.size() > 0) {
        for (P2PHKSignature p2PHKSignature : p2PHKSignatures) {
            if (p2PHKSignature != null) {
                size += SerializeUtils.sizeOfNulsData(p2PHKSignature);
            }
        }
```

```
        }
        if (scripts != null && scripts.size() > 0) {
            //
            size += NulsConstant.SIGN_HOLDER.length;
            //
            for (Script script : scripts) {
                if (script != null && script.getProgram() != null && script.getProgram().length > 0) {
                    size += SerializeUtils.sizeOfBytes(script.getProgram());
                }
            }
        }
        return size;
    }

    public List<P2PHKSignature> getP2PHKSignatures() {
        return p2PHKSignatures;
    }

    public void setP2PHKSignatures(List<P2PHKSignature> p2PHKSignatures) {
        this.p2PHKSignatures = p2PHKSignatures;
    }

    public List<Script> getScripts() {
        return scripts;
    }

    public void setScripts(List<Script> scripts) {
        this.scripts = scripts;
    }

    public int getSignersCount() {
        int count = 0;

        return count;
    }

}

68:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\service\KernelService.java
 */
```

```
package io.nuls.kernel.service;

import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.i18n.I18nUtils;
import io.nuls.kernel.lite.annotation.Service;
import io.nuls.kernel.model.NulsVersion;
import io.nuls.kernel.model.Result;

/**
 * @author: Niels Wang
 */
@Service
public class KernelService {

    public Result<NulsVersion> getVersion() {
        return Result.getSuccess().setData(NulsConfig.VERSION);
    }

    public Result setLanguage(String lang) {
        try {
            I18nUtils.setLanguage(lang);
        } catch (NulsException e) {
            return Result.getFailed(e.getErrorCode());
        }
        return Result.getSuccess();
    }
}

69:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\thread\BaseThread.java
 */
package io.nuls.kernel.thread;


import io.nuls.core.tools.log.Log;

/**
 * @author Niels
 */
public class BaseThread extends Thread {
    private short moduleId;
```

```
private String poolName;

public BaseThread() {
    super();
}

public BaseThread(Runnable target) {
    super(target);
}

public BaseThread(ThreadGroup group, Runnable target) {
    super(group, target);
}

public BaseThread(String name) {
    super(name);
}

public BaseThread(ThreadGroup group, String name) {
    super(group, name);
}

public BaseThread(Runnable target, String name) {
    super(target, name);
}

public BaseThread(ThreadGroup group, Runnable target, String name) {
    super(group, target, name);
}

public BaseThread(ThreadGroup group, Runnable target, String name, long stackSize) {
    super(group, target, name, stackSize);
}

@Override
public synchronized final void start() {
    this.beforeStart();
    super.start();
    this.afterStart();
}

protected void beforeStart() {
```

```
    //default do nothing
}

@Override
public final void run() {
    this.beforeRun();
    boolean ok = true;
    try {
        super.run();
    } catch (Exception e) {
        ok = false;
        runException(e);
    }
    if (ok) {
        this.afterRun();
    }

}

protected void runException(Exception e) {
    Log.error(e);
}

@Override
public final void interrupt() {
    this.beforeInterrupt();
    super.interrupt();
    this.afterInterrupt();
}

protected void afterStart() {
    //default do nothing
}

protected void afterRun() {
    //default do nothing
}

protected void beforeRun() {
    //default do nothing
}
```

```
    protected void afterInterrupt() {
        //default do nothing
    }

    protected void beforeInterrupt() {
        //default do nothing
    }

    public short getModuleId() {
        return moduleId;
    }

    public void setModuleId(short moduleId) {
        this.moduleId = moduleId;
    }

    public String getPoolName() {
        return poolName;
    }

    public void setPoolName(String poolName) {
        this.poolName = poolName;
    }
}

70:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\thread\cache\TaskTable.java
 */
package io.nuls.kernel.thread.cache;

import io.nuls.kernel.thread.BaseThread;

import java.util.*;
import java.util.concurrent.ThreadPoolExecutor;

/**
 * @author Niels
 */
public class TaskTable {

    private static final TaskTable INSTANCE = new TaskTable();
    /**
```

```
 * key  :  poolName
 * value:  pool
 */
private final Map<String, ThreadPoolExecutor> POOL_EXECUTOR_MAP = new HashMap<>();
/**
 * key  :  moduleId
 * value:  poolName
 */
private final Map<Short, Set<String>> MODULE_POOL_MAP = new HashMap<>();
/**
 * key  :  threadName
 * value:  thread
 */
private final Map<String, BaseThread> THREAD_MAP = new HashMap<>();
/**
 * key  :  moduleId
 * value:  threadName
 */
private final Map<Short, Set<String>> MODULE_THREAD_MAP = new HashMap<>();
/**
 * key  :  poolName
 * value:  Set<threadName>
 */
private final Map<String, Set<String>> POOL_THREAD_MAP = new HashMap<>();

private TaskTable() {
}

public static final TaskTable getInstance() {
    return INSTANCE;
}

public final void putPool(short moduleId, String poolName, ThreadPoolExecutor pool) {
    POOL_EXECUTOR_MAP.put(poolName, pool);
    Set<String> set = MODULE_POOL_MAP.get(moduleId);
    if (null == set) {
        set = new HashSet<>();
    }
    set.add(poolName);
    MODULE_POOL_MAP.put(moduleId, set);
}
```

```
    public final void putThread(short moduleId, String poolName, String threadName, BaseThread
thread) {
        THREAD_MAP.put(threadName, thread);
        Set<String> set1 = MODULE_THREAD_MAP.get(moduleId);
        if (null == set1) {
            set1 = new HashSet<>();
        }
        set1.add(threadName);
        MODULE_THREAD_MAP.put(moduleId, set1);
        Set<String> set = POOL_THREAD_MAP.get(poolName);
        if (null == set) {
            set = new HashSet<>();
        }
        set.add(threadName);
        POOL_THREAD_MAP.put(poolName, set);
    }

    public final BaseThread getThread(String threadName) {
        return THREAD_MAP.get(threadName);
    }

    public final ThreadPoolExecutor getPool(String poolName) {
        return POOL_EXECUTOR_MAP.get(poolName);
    }

    public final List<BaseThread> getThreadList(short moduleId) {
        Set<String> set = MODULE_THREAD_MAP.get(moduleId);
        if (null == set) {
            return null;
        }
        List<BaseThread> list = new ArrayList<>();
        for (String threadName : set) {
            list.add(THREAD_MAP.get(threadName));
        }
        return list;
    }

    public final List<BaseThread> getThreadList(String poolName) {
        Set<String> set = POOL_THREAD_MAP.get(poolName);
        if (null == set) {
            return null;
        }
```

```
    List<BaseThread> list = new ArrayList<>();
    for (String threadName : set) {
        list.add(THREAD_MAP.get(threadName));
    }
    return list;
}

public final List<ThreadPoolExecutor> getPoolList(short moduleId) {
    Set<String> set = MODULE_POOL_MAP.get(moduleId);
    if (null == set) {
        return null;
    }
    List<ThreadPoolExecutor> list = new ArrayList<>();
    for (String threadName : set) {
        list.add(POOL_EXECUTOR_MAP.get(threadName));
    }
    return list;
}

public void remove(short moduleId) {
    Set<String> threadNameSet = MODULE_THREAD_MAP.get(moduleId);
    for (String name : threadNameSet) {
        BaseThread thread = THREAD_MAP.get(name);
        thread.interrupt();
        THREAD_MAP.remove(name);
    }
    MODULE_THREAD_MAP.remove(moduleId);
    Set<String> poolNameSet = MODULE_POOL_MAP.get(moduleId);
    for (String name : poolNameSet) {
        ThreadPoolExecutor pool = POOL_EXECUTOR_MAP.get(name);
        pool.shutdown();
        POOL_EXECUTOR_MAP.remove(name);
        POOL_THREAD_MAP.remove(name);
    }
    MODULE_POOL_MAP.remove(moduleId);

}

public void removeThread(short moduleId, String threadName) {
    THREAD_MAP.remove(threadName);
    Set<String> set = MODULE_THREAD_MAP.get(moduleId);
    set.remove(threadName);
```

```
        for (Set<String> tset : POOL_THREAD_MAP.values()) {
            tset.remove(threadName);
        }
    }
}

71:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\thread\manager\NulsThreadFactory.java
 */
package io.nuls.kernel.thread.manager;

import io.nuls.kernel.thread.BaseThread;

import java.util.concurrent.ThreadFactory;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * @author Niels
 */
public class NulsThreadFactory implements ThreadFactory {
    private final short moduleId;
    private final String poolName;
    private AtomicInteger threadNo = new AtomicInteger(1);

    public NulsThreadFactory(short moduleId, String poolName) {
        this.poolName = poolName;
        this.moduleId = moduleId;
    }

    @Override
    public final Thread newThread(Runnable r) {
        String threadName;
        if (threadNo.get() == 1) {
            threadNo.incrementAndGet();
            threadName = "[" + poolName + "]";
        } else {
            threadName = "[" + poolName + "-" + threadNo.getAndIncrement() + "]";
        }
        BaseThread newThread = new BaseThread(r, threadName);
        newThread.setModuleId(moduleId);
        newThread.setPoolName(poolName);
        newThread.setDaemon(true);
```

```
        if (newThread.getPriority() != Thread.NORM_PRIORITY) {
            newThread.setPriority(Thread.NORM_PRIORITY);
        }
        TaskManager.putThread(moduleId, poolName, threadName, newThread);
        return newThread;
    }

    public String getPoolName() {
        return poolName;
    }

    public short getModuleId() {
        return moduleId;
    }
}

72:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\thread\manager\TaskManager.java
 */
package io.nuls.kernel.thread.manager;

import io.nuls.core.tools.aop.AopUtils;
import io.nuls.core.tools.log.Log;
import io.nuls.kernel.thread.BaseThread;
import io.nuls.kernel.thread.cache.TaskTable;

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

/**
 * @author Niels
 */
public class TaskManager {

    private static final TaskTable THREAD_DATA_CACHE = TaskTable.getInstance();

    private static final String TEMPORARY_THREAD_POOL_NAME = "temporary";
    private static final int TEMPORARY_THREAD_POOL_COUNT = 4;
    private static final int TEMPORARY_THREAD_POOL_QUEUE_SIZE = 1000;
    private static final ThreadPoolExecutor TEMPORARY_THREAD_POOL;

    /**
```

```
     * Initializing a temporary thread pool
     */
    static {
        TEMPORARY_THREAD_POOL =
createThreadPool(TEMPORARY_THREAD_POOL_COUNT,
TEMPORARY_THREAD_POOL_QUEUE_SIZE, new NulsThreadFactory((short) 0,
TEMPORARY_THREAD_POOL_NAME));
    }

    public static final void putThread(short moduleId, String poolName, String threadName,
BaseThread newThread) {
        THREAD_DATA_CACHE.putThread(moduleId, poolName, threadName, newThread);
    }

    public static final ThreadPoolExecutor createThreadPool(int threadCount, int queueSize,
NulsThreadFactory factory) {
        if (threadCount == 0) {
            throw new RuntimeException("thread count cannot be 0!");
        }
        if (factory == null) {
            throw new RuntimeException("thread factory cannot be null!");
        }
        Class[] paramClasses = new Class[]{int.class, int.class, long.class, TimeUnit.class,
BlockingQueue.class, ThreadFactory.class};
        Object[] paramArgs = null;
        if (queueSize > 0) {
            paramArgs = new Object[]{threadCount, threadCount, 0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<>(queueSize), factory};
        } else {
            paramArgs = new Object[]{threadCount, threadCount, 0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<>(), factory};
        }
        ThreadPoolExecutor pool = AopUtils.createProxy(ThreadPoolExecutor.class, paramClasses,
paramArgs, new ThreadPoolInterceiptor());
        THREAD_DATA_CACHE.putPool(factory.getModuleId(), factory.getPoolName(), pool);
        return pool;
    }

    public static final ScheduledThreadPoolExecutor createScheduledThreadPool(int threadCount,
NulsThreadFactory factory) {
        if (factory == null) {
            throw new RuntimeException("thread factory cannot be null!");
```

```
        }
        ScheduledThreadPoolExecutor pool =
AopUtils.createProxy(ScheduledThreadPoolExecutor.class, new Class[]{int.class,
ThreadFactory.class}, new Object[]{threadCount, factory}, new ThreadPoolInterceiptor());
        THREAD_DATA_CACHE.putPool(factory.getModuleId(), factory.getPoolName(), pool);
        return pool;
    }

    public static final ScheduledThreadPoolExecutor
createScheduledThreadPool(NulsThreadFactory factory) {
        return createScheduledThreadPool(1, factory);
    }

    public static final void asynExecuteRunnable(Runnable runnable) {
        if (null == runnable) {
            throw new RuntimeException("runnable is null");
        }
        if (TEMPORARY_THREAD_POOL == null) {
            throw new RuntimeException("temporary thread pool not initialized yet");
        }
        BlockingQueue<Runnable> blockingQueue = TEMPORARY_THREAD_POOL.getQueue();
        if (blockingQueue.size() > 200) {
            Log.info("Task Queue 100 Size Warning!!! Task info is " + runnable.toString());
        }
        TEMPORARY_THREAD_POOL.execute(runnable);
        int i = TEMPORARY_THREAD_POOL.getQueue().size();
        if (1 > 10) {
            System.out.println("thread pool size:" + i);
        }
    }

    public static final void createAndRunThread(short moduleId, String threadName, Runnable
runnable) {
        createAndRunThread(moduleId, threadName, runnable, true);
    }

    public static final void createAndRunThread(short moduleId, String threadName, Runnable
runnable, boolean deamon) {
        NulsThreadFactory factory = new NulsThreadFactory(moduleId, threadName);
        Thread thread = factory.newThread(runnable);
        thread.setDaemon(deamon);
        thread.start();
```

```
    }

    public static final List<BaseThread> getThreadList(short moduleId) {
        return THREAD_DATA_CACHE.getThreadList(moduleId);
    }

    public static void shutdownByModuleId(short moduleId) {
        List<ThreadPoolExecutor> poolList = THREAD_DATA_CACHE.getPoolList(moduleId);
        if (null != poolList) {
            for (ThreadPoolExecutor pool : poolList) {
                pool.shutdown();
            }
        }
        List<BaseThread> threadList = THREAD_DATA_CACHE.getThreadList(moduleId);
        if (null != threadList) {
            for (Thread thread : threadList) {
                if (thread.getState() == Thread.State.RUNNABLE) {
                    thread.interrupt();
                }
            }
        }
        THREAD_DATA_CACHE.remove(moduleId);
    }

    public static BaseThread getThread(String threadName) {
        BaseThread thread = THREAD_DATA_CACHE.getThread(threadName);
        return thread;
    }

    public static void stopThread(short moduleId, String threadName) {
        BaseThread thread = THREAD_DATA_CACHE.getThread(threadName);
        if (thread.getState() == Thread.State.RUNNABLE) {
            thread.interrupt();
        }
        THREAD_DATA_CACHE.removeThread(moduleId, threadName);

    }
}

73:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\thread\manager\ThreadPoolInterceiptor.java
 */
```

```
package io.nuls.kernel.thread.manager;

import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

/**
 * @author Niels
 */
public class ThreadPoolInterceiptor implements MethodInterceptor {
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws
Throwable {
        return proxy.invokeSuper(obj, args);
    }
}

74:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\AddressTool.java
 */

package io.nuls.kernel.utils;

import io.nuls.core.tools.array.ArraysTool;
import io.nuls.core.tools.crypto.Base58;
import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.model.Address;



/**
 * @author: Niels Wang
 */
public class AddressTool {

    public static byte[] getAddress(String addressString) {
        byte[] bytes;
```

```
        try {
            bytes = Base58.decode(addressString);
        } catch (Exception e) {
            Log.error(e);
            throw new NulsRuntimeException(e);
        }
        byte[] result = new byte[Address.ADDRESS_LENGTH];
        System.arraycopy(bytes, 0, result, 0, Address.ADDRESS_LENGTH);
        return result;
    }

    public static byte[] getAddress(byte[] publicKey) {
        if (publicKey == null) {
            return null;
        }
        byte[] hash160 = SerializeUtils.sha256hash160(publicKey);
        Address address = new Address(NulsContext.DEFAULT_CHAIN_ID,
NulsContext.DEFAULT_ADDRESS_TYPE, hash160);
        return address.getAddressBytes();
    }

    private static byte getXor(byte[] body) {
        byte xor = 0x00;
        for (int i = 0; i < body.length; i++) {
            xor ^= body[i];
        }
        return xor;
    }

    public static boolean validAddress(String address) {
        if (StringUtils.isBlank(address)) {
            return false;
        }
        byte[] bytes;
        try {
            bytes = Base58.decode(address);
            if (bytes.length != Address.ADDRESS_LENGTH + 1) {
                return false;
            }
        } catch (NulsException e) {
            return false;
        } catch (Exception e) {
```

```
            return false;
        }
        NulsByteBuffer byteBuffer = new NulsByteBuffer(bytes);
        short chainId;
        byte type;
        try {
            chainId = byteBuffer.readShort();
            type = byteBuffer.readByte();
        } catch (NulsException e) {
            Log.error(e);
            return false;
        }
        if (NulsContext.DEFAULT_CHAIN_ID != chainId) {
            return false;
        }
        if (NulsContext.MAIN_NET_VERSION <= 1 && NulsContext.DEFAULT_ADDRESS_TYPE !=
type) {
            return false;
        }
        if (NulsContext.DEFAULT_ADDRESS_TYPE != type &&
NulsContext.CONTRACT_ADDRESS_TYPE != type && NulsContext.P2SH_ADDRESS_TYPE !=
type) {
            return false;
        }
        try {
            checkXOR(bytes);
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    public static boolean validNormalAddress(byte[] bytes) {
        if (null == bytes || bytes.length != Address.ADDRESS_LENGTH) {
            return false;
        }
        NulsByteBuffer byteBuffer = new NulsByteBuffer(bytes);
        short chainId;
        byte type;
        try {
            chainId = byteBuffer.readShort();
            type = byteBuffer.readByte();
```

```
    } catch (NulsException e) {
        Log.error(e);
        return false;
    }
    if (NulsContext.DEFAULT_CHAIN_ID != chainId) {
        return false;
    }
    if (NulsContext.DEFAULT_ADDRESS_TYPE != type) {
        return false;
    }
    return true;
}

public static boolean validContractAddress(byte[] addressBytes) {
    if (addressBytes == null) {
        return false;
    }
    if (addressBytes.length != Address.ADDRESS_LENGTH) {
        return false;
    }
    NulsByteBuffer byteBuffer = new NulsByteBuffer(addressBytes);
    short chainId;
    byte type;
    try {
        chainId = byteBuffer.readShort();
        type = byteBuffer.readByte();
    } catch (NulsException e) {
        Log.error(e);
        return false;
    }
    if (NulsContext.DEFAULT_CHAIN_ID != chainId) {
        return false;
    }
    if (NulsContext.CONTRACT_ADDRESS_TYPE != type) {
        return false;
    }
    return true;
}


public static void checkXOR(byte[] hashs) {
    byte[] body = new byte[Address.ADDRESS_LENGTH];
```

```
    System.arraycopy(hashs, 0, body, 0, Address.ADDRESS_LENGTH);

    byte xor = 0x00;
    for (int i = 0; i < body.length; i++) {
        xor ^= body[i];
    }
    byte[] sign = new byte[1];
    System.arraycopy(hashs, Address.ADDRESS_LENGTH, sign, 0, 1);

    if (xor != hashs[Address.ADDRESS_LENGTH]) {
        throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
    }
}

public static String getStringAddressByBytes(byte[] addressBytes) {
    byte[] bytes = ArraysTool.concatenate(addressBytes, new byte[]{getXor(addressBytes)});
    return Base58.encode(bytes);
}



public static boolean checkPublicKeyHash(byte[] address, byte[] pubKeyHash) {

    if (address == null || pubKeyHash == null) {
        return false;
    }
    int pubKeyHashLength = pubKeyHash.length;
    if (address.length != Address.ADDRESS_LENGTH || pubKeyHashLength != 20) {
        return false;
    }
    for (int i = 0; i < pubKeyHashLength; i++) {
        if (pubKeyHash[i] != address[i + 3]) {
            return false;
        }
    }
    return true;
}

public static boolean isPay2ScriptHashAddress(byte[] addr) {
    if (addr != null && addr.length > 3) {
        return addr[2] == NulsContext.P2SH_ADDRESS_TYPE;
    }
```

```
        return false;
    }

    public static boolean isPackingAddress(String address) {
        if (StringUtils.isBlank(address)) {
            return false;
        }
        byte[] bytes;
        try {
            bytes = Base58.decode(address);
            if (bytes.length != Address.ADDRESS_LENGTH + 1) {
                return false;
            }
        } catch (NulsException e) {
            return false;
        } catch (Exception e) {
            return false;
        }
        NulsByteBuffer byteBuffer = new NulsByteBuffer(bytes);
        short chainId;
        byte type;
        try {
            chainId = byteBuffer.readShort();
            type = byteBuffer.readByte();
        } catch (NulsException e) {
            Log.error(e);
            return false;
        }
        if (NulsContext.DEFAULT_CHAIN_ID != chainId) {
            return false;
        }
        if (NulsContext.DEFAULT_ADDRESS_TYPE != type) {
            return false;
        }
        try {
            checkXOR(bytes);
        } catch (Exception e) {
            return false;
        }
        return true;
    }
}
```

75:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\utils\ByteArrayWrapper.java

```
package io.nuls.kernel.utils;

import static java.util.Objects.requireNonNull;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/6/28
 */
public class ByteArrayWrapper implements Comparable<ByteArrayWrapper> {

    private final byte[] data;
    private final int offset;
    private final int length;

    private int hash;

    public ByteArrayWrapper(byte[] data) {
        requireNonNull(data, "array is null");
        this.data = data;
        this.offset = 0;
        this.length = data.length;
    }

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

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

        ByteArrayWrapper byteArrayWrapper = (ByteArrayWrapper) o;
```

```
    // do lengths match
    if (length != byteArrayWrapper.length) {
        return false;
    }

    // if arrays have same base offset, some optimizations can be taken...
    if (offset == byteArrayWrapper.offset && data == byteArrayWrapper.data) {
        return true;
    }
    for (int i = 0; i < length; i++) {
        if (data[offset + i] != byteArrayWrapper.data[byteArrayWrapper.offset + i]) {
            return false;
        }
    }
    return true;
}

@Override
public int hashCode() {
    if (hash != 0) {
        return hash;
    }

    int result = length;
    for (int i = offset; i < offset + length; i++) {
        result = 31 * result + data[i];
    }
    if (result == 0) {
        result = 1;
    }
    hash = result;
    return hash;
}

@Override
public int compareTo(ByteArrayWrapper that) {
    if (this == that) {
        return 0;
    }
    if (this.data == that.data && length == that.length && offset == that.offset) {
        return 0;
    }
```

```
        int minLength = Math.min(this.length, that.length);
        for (int i = 0; i < minLength; i++) {
            int thisByte = 0xFF & this.data[this.offset + i];
            int thatByte = 0xFF & that.data[that.offset + i];
            if (thisByte != thatByte) {
                return (thisByte) - (thatByte);
            }
        }
        return this.length - that.length;
    }
}

76:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\CommandBuilder.java
 */
package io.nuls.kernel.utils;

import io.nuls.core.tools.str.StringUtils;

/**
 * @author: Niels Wang
 */
public class CommandBuilder {
    private StringBuilder builder = new StringBuilder();
    private static final String LINE_SEPARATOR = "line.separator";
    private int i = 0;

    public CommandBuilder newLine(String content) {
        if (StringUtils.isBlank(content)) {
            return this.newLine();
        }
        builder.append(content).append(System.getProperty(LINE_SEPARATOR));
        if (i++ == 0) {
            this.newLine("\tOPTIONS:");
        }
        return this;
    }

    public CommandBuilder newLine() {
        builder.append(System.getProperty(LINE_SEPARATOR));
        return this;
```

```
    }

    @Override
    public String toString() {
        if (i == 2) {
            this.newLine("\tnone");
        }
        return builder.toString();
    }
}

77:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\CommandHelper.java
 */

package io.nuls.kernel.utils;

import io.nuls.core.tools.json.JSONUtils;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.RpcClientResult;
import jline.console.ConsoleReader;

import java.io.IOException;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

/**
 * @author: Charlie
 */
public class CommandHelper {

    public static boolean checkArgsIsNull(String... args) {
        for (String arg : args) {
            if (arg == null || arg.trim().length() == 0) {
                return false;
            }
        }
        return true;
    }
```

```
    //
//    /**
//     *
//     * @return
//     */
    public static String getNewPwd() {
        System.out.print("Please enter the new password(8-20 characters, the combination of letters
and numbers).\nEnter your new password:");
        ConsoleReader reader = null;
        try {
            reader = new ConsoleReader();
            String pwd = null;
            do {
                pwd = reader.readLine('*');
                if (!StringUtils.validPassword(pwd)) {
                    System.out.print("The password is invalid, (8-20 characters, the combination of letters
and numbers) .\nReenter the new password: ");
                }
            } while (!StringUtils.validPassword(pwd));
            return pwd;
        } catch (IOException e) {
            return null;
        } finally {
            try {
                if (!reader.delete()) {
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }


    //    /**
//     *
//     * @param newPwd
//     */
    public static void confirmPwd(String newPwd) {
        System.out.print("Please confirm new password:");
        ConsoleReader reader = null;
        try {
```

```
            reader = new ConsoleReader();
            String confirmed = null;
            do {
                confirmed = reader.readLine('*');
                if (!newPwd.equals(confirmed)) {
                    System.out.print("Password confirmation doesn't match the password.\nConfirm new
password: ");
                }
            } while (!newPwd.equals(confirmed));
        } catch (IOException e) {

        } finally {
            try {
                if (!reader.delete()) {
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    //
//    /**
//     * ,
//     *
//     * @return
//     */
    public static String getPwd() {
        return getPwd(null);
    }

    //    /**
//     * ,
//     * @param prompt
//     * @return
//     */
    public static String getPwd(String prompt) {
        if (StringUtils.isBlank(prompt)) {
            prompt = "Please enter the password.\nEnter your password:";
        }
        System.out.print(prompt);
```

```
        ConsoleReader reader = null;
        try {
            reader = new ConsoleReader();
            String npwd = null;
            do {
                npwd = reader.readLine('*');
                if ("".equals(npwd)) {
                    System.out.print("The password is required.\nEnter your password:");
                }
            } while ("".equals(npwd));
            return npwd;
        } catch (IOException e) {
            return null;
        } finally {
            try {
                if (!reader.delete()) {
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    //
//    /**
//     * ,
//     * @param prompt
//     * @return
//     */
    public static String getPwdOptional(String prompt) {
        if (StringUtils.isBlank(prompt)) {
            prompt = "Please enter the password (password is between 8 and 20 inclusive of numbers
and letters), " +
                    "If you do not want to set a password, return directly.\nEnter your password:";
        }
        System.out.print(prompt);
        ConsoleReader reader = null;
        try {
            reader = new ConsoleReader();
            String npwd = null;
            do {
```

```
            npwd = reader.readLine('*');
            if (!"".equals(npwd) && !StringUtils.validPassword(npwd)) {
                System.out.print("Password invalid, password is between 8 and 20 inclusive of
numbers and letters.\nEnter your password:");
            }
        } while (!"".equals(npwd) && !StringUtils.validPassword(npwd));
        return npwd;
    } catch (IOException e) {
        return null;
    } finally {
        try {
            if (!reader.delete()) {
                reader.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

/**
 * ,
 *
 *
 * @return
 */
public static String getPwdOptional() {
    return getPwdOptional(null);
}


public static Long getLongAmount(String arg) {
    Na na = null;
    try {
        na = Na.parseNuls(arg);
        return na.getValue();
    } catch (Exception e) {
        return null;
    }
}

public static String naToNuls(Object object) {
```

```
    if (null == object) {
        return null;
    }
    Long na = null;
    if (object instanceof Long) {
        na = (Long) object;
    } else if (object instanceof Integer) {
        na = ((Integer) object).longValue();
    } else {
        return null;
    }
    return (Na.valueOf(na)).toText();
}

public static String txTypeExplain(Integer type) {
    if (null == type) {
        return null;
    }
    switch (type) {
        case 1:
            return "coinbase";
        case 2:
            return "transfer";
        case 3:
            return "account_alias";
        case 4:
            return "register_agent";
        case 5:
            return "join_consensus";
        case 6:
            return "cancel_deposit";
        case 7:
            return "yellow_punish";
        case 8:
            return "red_punish";
        case 9:
            return "stop_agent";
        default:
            return type.toString();
    }
}
```

```
    public static String consensusExplain(Integer status) {
        if (null == status) {
            return null;
        }
        switch (status) {
            case 0:
                return "unconsensus";
            case 1:
                return "consensus";

            default:
                return status.toString();
        }
    }

    public static String statusConfirmExplain(Integer status) {
        if (null == status) {
            return null;
        }
        switch (status) {
            case 0:
                return "confirm";
            case 1:
                return "unConfirm";

            default:
                return status.toString();
        }
    }


    //    /**
//     *
//     * 1.,
//     * 2.,
//     *
//     * @param address
//     * @param restFul
//     * @return RpcClientResult
//     */
    public static RpcClientResult getPassword(String address, RestFulUtils restFul) {
        return getPassword(address, restFul, null);
```

```
    }

    //    /**
//     *
//     * 1.,
//     * 2.,
//     *
//     * @param address
//     * @param restFul
//     * @param prompt
//     * @return RpcClientResult
//     */
    public static RpcClientResult getPassword(String address, RestFulUtils restFul, String prompt) {
        if (StringUtils.isBlank(address)) {
            return RpcClientResult.getFailed("address is wrong");
        }
        RpcClientResult result = restFul.get("/account/encrypted/" + address, null);
        if (result.isSuccess()) {
            RpcClientResult rpcClientResult = new RpcClientResult();
            rpcClientResult.setSuccess(true);
            if (result.dataToBooleanValue()) {
                String pwd = getPwd(prompt);
                rpcClientResult.setData(pwd);
            }
            return rpcClientResult;
        }
        return result;

    }


    private static String getArgsJson() {
        String prompt = "Please enter the arguments according to the arguments structure(eg. \"a\",2,[\"c\",4],\"\",\"e\" or \"'a',2,['c',4],'','e'\")," +
                "\nIf this method has no arguments(Refer to the command named \"getcontractinfo\" for the arguments structure of the method.), return directly.\nEnter the arguments:";
        System.out.print(prompt);
        ConsoleReader reader = null;
        try {
            reader = new ConsoleReader();
            String args = reader.readLine();
            if(StringUtils.isNotBlank(args)) {
```

```
            args = "[" + args + "]";
        }
        return args;
    } catch (IOException e) {
        return null;
    } finally {
        try {
            if (!reader.delete()) {
                reader.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

private static Object[] parseArgsJson(String argsJson) {
    if(StringUtils.isBlank(argsJson)) {
        return new Object[0];
    }
    try {
        List<Object> list = JSONUtils.json2pojo(argsJson, ArrayList.class);
        return list.toArray();
    } catch (Exception e) {
        e.fillInStackTrace();
        return null;
    }
}

public static RpcClientResult getContractCallArgsJson() {
    RpcClientResult rpcClientResult = new RpcClientResult();
    rpcClientResult.setSuccess(true);
    try {
        Object[] argsObj;
        //
        String argsJson = getArgsJson();
        argsObj = parseArgsJson(argsJson);
        rpcClientResult.setData(argsObj);
    } catch (Exception e) {
        e.printStackTrace();
        rpcClientResult.setSuccess(false);
    }
```

```
        return rpcClientResult;
    }


    public static String tokenRecovery(String amount, Integer decimals) {
        if(StringUtils.isBlank(amount) || decimals == null) {
            return null;
        }
        return new BigDecimal(amount).divide(BigDecimal.TEN.pow(decimals)).toPlainString();
    }
}

78:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\MappedBufferCleanUtil.java
 */
package io.nuls.kernel.utils;


import sun.nio.ch.FileChannelImpl;

import java.lang.reflect.Method;
import java.nio.MappedByteBuffer;

/**
 * @author Niels
 * @date 2017/9/21
 */
public class MappedBufferCleanUtil {


    public static void clean(final Object buffer) {
        if (null == buffer) {
            return;
        }
        try {
            //unmap
            Method m = FileChannelImpl.class.getDeclaredMethod("unmap",
                    MappedByteBuffer.class);
            m.setAccessible(true);
            m.invoke(FileChannelImpl.class, buffer);
            //Another way of implementation
            //            Method getCleanerMethod = buffer.getClass().getMethod("cleaner", new
```

```
Class[0]);
            //          getCleanerMethod.setAccessible(true);
            //          sun.misc.Cleaner cleaner = (sun.misc.Cleaner) getCleanerMethod.invoke(buffer,
new Object[0]);
            //          cleaner.clean();
            //          getCleanerMethod.setAccessible(false);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}

79:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\NulsByteBuffer.java
 */
package io.nuls.kernel.utils;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.model.BaseNulsData;
import io.nuls.kernel.model.NulsDigestData;
import io.nuls.kernel.model.NulsSignData;
import io.nuls.kernel.model.Transaction;

import java.io.UnsupportedEncodingException;
import java.util.Arrays;

import static io.nuls.core.tools.str.StringUtils.EMPTY;

/**
 * @author Niels
 */
public class NulsByteBuffer {

    private final byte[] payload;

    private int cursor;
```

```
public NulsByteBuffer(byte[] bytes) {
    this(bytes, 0);
}

public NulsByteBuffer(byte[] bytes, int cursor) {
    if (null == bytes || bytes.length == 0 || cursor < 0) {
        throw new NulsRuntimeException(KernelErrorCode.PARAMETER_ERROR);
    }
    this.payload = bytes;
    this.cursor = cursor;
}

public long readUint32LE() throws NulsException {
    try {
        long u = SerializeUtils.readUint32LE(payload, cursor);
        cursor += 4;
        return u;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}

public int readUint16() throws NulsException {
    try {
        int val = SerializeUtils.readUint16LE(payload, cursor);
        cursor += 2;
        return val;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}

public int readInt32() throws NulsException {
    try {
        int u = SerializeUtils.readInt32LE(payload, cursor);
        cursor += 4;
        return u;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}
```

```
public long readUint32() throws NulsException {
    try {
        long val = SerializeUtils.readUint32LE(payload, cursor);
        cursor += 4;
        return val;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}

public long readInt64() throws NulsException {
    try {
        long u = SerializeUtils.readInt64LE(payload, cursor);
        cursor += 8;
        return u;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}


public long readVarInt() throws NulsException {
    return readVarInt(0);
}

public long readVarInt(int offset) throws NulsException {
    try {
        VarInt varint = new VarInt(payload, cursor + offset);
        cursor += offset + varint.getOriginalSizeInBytes();
        return varint.value;
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
    }
}

public byte readByte() throws NulsException {
    try {
        byte b = payload[cursor];
        cursor += 1;
        return b;
    } catch (IndexOutOfBoundsException e) {
```

```
            throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
        }
    }

    public byte[] readBytes(int length) throws NulsException {
        try {
            byte[] b = new byte[length];
            System.arraycopy(payload, cursor, b, 0, length);
            cursor += length;
            return b;
        } catch (IndexOutOfBoundsException e) {
            throw new NulsException(KernelErrorCode.DATA_PARSE_ERROR, e);
        }
    }

    public byte[] readByLengthByte() throws NulsException {
        long length = this.readVarInt();
        if (length == 0) {
            return null;
        }
        return readBytes((int) length);
    }

    public boolean readBoolean() throws NulsException {
        byte b = readByte();
        return 1 == b;
    }

    public NulsDigestData readHash() throws NulsException {
        return this.readNulsData(new NulsDigestData());
    }

    public void resetCursor() {
        this.cursor = 0;
    }

    public short readShort() throws NulsException {
        byte[] bytes = this.readBytes(2);
        if (null == bytes) {
            return 0;
        }
        return SerializeUtils.bytes2Short(bytes);
```

```
    }

    public String readString() throws NulsException {
        try {
            byte[] bytes = this.readByLengthByte();
            if (null == bytes) {
                return EMPTY;
            }
            return new String(bytes, NulsConfig.DEFAULT_ENCODING);
        } catch (UnsupportedEncodingException e) {
            Log.error(e);
            throw new NulsException(e);
        }

    }

    public double readDouble() throws NulsException {
        byte[] bytes = this.readBytes(8);
        if (null == bytes) {
            return 0;
        }
        return SerializeUtils.bytes2Double(bytes);
    }

    public boolean isFinished() {
        return this.payload.length == cursor;
    }

//    public byte[] getPayloadByCursor() {
//        byte[] bytes = new byte[payload.length - cursor];
//        System.arraycopy(this.payload, cursor, bytes, 0, bytes.length);
//        return bytes;
//    }

    public byte[] getPayload() {
        return payload;
    }

    public <T extends BaseNulsData> T readNulsData(T nulsData) throws NulsException {
        if (payload == null) {
            return null;
        }
```

```
    int length = payload.length - cursor;
    if (length <= 0) {
        return null;
    }
    if (length >= 4) {
        byte[] byte4 = new byte[4];
        System.arraycopy(payload, cursor, byte4, 0, 4);
        if (Arrays.equals(NulsConstant.PLACE_HOLDER, byte4)) {
            cursor += 4;
            return null;
        }
    }
    nulsData.parse(this);
    return nulsData;
}

public NulsSignData readSign() throws NulsException {
    return this.readNulsData(new NulsSignData());
}

public long readUint48() {
    long value = (payload[cursor + 0] & 0xffL) |
            ((payload[cursor + 1] & 0xffL) << 8) |
            ((payload[cursor + 2] & 0xffL) << 16) |
            ((payload[cursor + 3] & 0xffL) << 24) |
            ((payload[cursor + 4] & 0xffL) << 32) |
            ((payload[cursor + 5] & 0xffL) << 40);
    //todo
    cursor += 6;
    if (value == 281474976710655L) {
        return -1L;
    }
    return value;
}

public Transaction readTransaction() throws NulsException {
    try {
        return TransactionManager.getInstance(this);
    } catch (Exception e) {
        Log.error(e);
        throw new NulsException(e);
    }
```

```
    }

    public int getCursor() {
        return cursor;
    }

    public void setCursor(int cursor) {
        this.cursor = cursor;
    }
}

80:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\NulsOutputStreamBuffer.java
 */
package io.nuls.kernel.utils;

import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.model.BaseNulsData;

import java.io.IOException;
import java.io.OutputStream;

/**
 * @author Niels
 */
public class NulsOutputStreamBuffer {

    private final OutputStream out;

    public NulsOutputStreamBuffer(OutputStream out) {
        this.out = out;
    }

    public void write(byte[] bytes) throws IOException {
        out.write(bytes);
    }

    public void write(int val) throws IOException {
```

```
    out.write(val);
}

public void writeVarInt(int val) throws IOException {
    out.write(new VarInt(val).encode());
}

public void writeVarInt(long val) throws IOException {
    out.write(new VarInt(val).encode());
}


public void writeBytesWithLength(byte[] bytes) throws IOException {
    if (null == bytes || bytes.length == 0) {
        out.write(new VarInt(0).encode());
    } else {
        out.write(new VarInt(bytes.length).encode());
        out.write(bytes);
    }
}

public void writeBoolean(boolean val) throws IOException {
    out.write(val ? 1 : 0);
}

public void writeShort(short val) throws IOException {
    SerializeUtils.int16ToByteStreamLE(val, out);
}

public void writeUint16(int val) throws IOException {
    SerializeUtils.uint16ToByteStreamLE(val, out);
}

public void writeUint32(long val) throws IOException {
    SerializeUtils.uint32ToByteStreamLE(val, out);
}

public void writeInt64(long val) throws IOException {
    SerializeUtils.int64ToByteStreamLE(val, out);
}

public void writeDouble(double val) throws IOException {
```

```
        out.write(SerializeUtils.double2Bytes(val));
    }

    public void writeString(String val) {
        if (StringUtils.isBlank(val)) {
            try {
                out.write(new VarInt(0).encode());
            } catch (IOException e) {
                Log.error(e);
                throw new NulsRuntimeException(e);
            }
            return;
        }
        try {
            this.writeBytesWithLength(val.getBytes(NulsConfig.DEFAULT_ENCODING));
        } catch (IOException e) {
            Log.error(e);
            throw new NulsRuntimeException(e);
        }
    }

    public void writeNulsData(BaseNulsData data) throws IOException {
        if (null == data) {
            write(NulsConstant.PLACE_HOLDER);
        } else {
            this.write(data.serialize());
        }
    }

    public void writeUint48(long time) throws IOException {
        byte[] bytes = SerializeUtils.uint48ToBytes(time);
        this.write(bytes);
    }
}

81:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\entity\PersistentQueue.java
 */
package io.nuls.kernel.utils.queue.entity;


import io.nuls.kernel.utils.queue.fqueue.entity.FQueue;
```

```
import java.io.File;
import java.io.IOException;
import java.net.URLDecoder;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Fqueue
 * Created by Niels on 2017/9/20.
 */
public class PersistentQueue {

    private final String queueName;
    private final long maxSize;
    private FQueue queue = null;
    /**
     * Lock held by take, poll, etc
     */
    private final ReentrantLock takeLock = new ReentrantLock();

    /**
     * Wait queue for waiting takes
     */
    private final Condition notEmpty = takeLock.newCondition();

    /**
     *
     *
     * @param queueName
     * @param maxSize   fileLimitLength
     */
    public PersistentQueue(String queueName, long maxSize) throws Exception {
        this.queueName =
URLDecoder.decode(PersistentQueue.class.getClassLoader().getResource("").getPath() +
"/data/queue/" + queueName, "UTF-8");
        this.maxSize = maxSize;
        this.queue = new FQueue(this.queueName, maxSize);
    }

    public void offer(byte[] t) {
        if (null == t || this.queue == null) {
```

```
            return;
        }
        this.queue.offer(t);
        if (size() > 0) {
            signalNotEmpty();
        }
    }

    @Deprecated
    public void remove(byte[] item) {
        this.queue.remove(item);
    }

    public byte[] poll() {
        return this.queue.poll();
    }

    public byte[] take() throws InterruptedException {
        byte[] x;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lockInterruptibly();
        try {
            while (size() == 0) {
                notEmpty.await();
            }
            x = poll();
            if (null == x) {
                return take();
            }
            if (size() > 0) {
                notEmpty.signal();
            }
        } finally {
            takeLock.unlock();
        }
        return x;
    }

    private void signalNotEmpty() {
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
```

```
            notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
    }

    public long size() {
        return this.queue.size();
    }

    public void close() throws IOException {
        this.queue.close();
    }

    public void clear() {
        this.queue.clear();
    }

    public void distroy() throws IOException {
        if (null == queue) {
            return;
        }
        this.queue.close();
        File file = new File(this.queueName);
        this.deleteFile(file);
    }

    private void deleteFile(File file) {
        if (null != file && file.exists()) {//
            if (file.isFile()) {//
                boolean b = file.delete();//
//                Log.debug(":" + file.getPath() + "" + b);
            } else if (file.isDirectory()) {//
                File[] files = file.listFiles();// files[];
                for (File f : files) {//
                    this.deleteFile(f);//
                }
                boolean b = file.delete();//
//                Log.debug(":" + file.getPath() + "" + b);
            }
        }
    }
```

```
}

82:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\fqueue\entity\FQueue.java
 */
package io.nuls.kernel.utils.queue.fqueue.entity;


import io.nuls.core.tools.log.Log;

import java.io.File;
import java.io.IOException;
import java.io.Serializable;
import java.util.AbstractQueue;
import java.util.Iterator;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 *
 *
 * @author opensource
 */
public class FQueue extends AbstractQueue<byte[]> implements Serializable {
    private static final long serialVersionUID = -1L;

    private FSQueue fsQueue = null;
    private Lock lock = new ReentrantReadWriteLock().writeLock();

    public FQueue(String path) throws IOException {
        try {
            fsQueue = new FSQueue(path);
        } catch (Exception e) {
            Log.error(e);
        }
    }

    /**
     *
     *
     * @param path
```

```
     * @param entityLimitLength
     */
    public FQueue(String path, long entityLimitLength) throws Exception {
        fsQueue = new FSQueue(path, entityLimitLength);
    }

    public FQueue(File dir) throws Exception {
        fsQueue = new FSQueue(dir);
    }

    /**
     *
     *
     * @param dir
     * @param entityLimitLength
     */
    public FQueue(File dir, int entityLimitLength) throws Exception {
        fsQueue = new FSQueue(dir, entityLimitLength);
    }

    @Override
    public Iterator<byte[]> iterator() {
        throw new UnsupportedOperationException("iterator Unsupported now");
    }

    @Override
    public int size() {
        return fsQueue.getQueueSize();
    }

    @Override
    public boolean offer(byte[] e) {
        try {
            lock.lock();
            fsQueue.add(e);
            return true;
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        } catch (Exception ex) {
        } finally {
            lock.unlock();
        }
```

```
    return false;
}

@Override
public byte[] peek() {
    try {
        lock.lock();
        return fsQueue.readNext();
    } catch (IOException ex) {
        return null;
    } catch (Exception ex) {
        return null;
    } finally {
        lock.unlock();
    }
}

@Override
public byte[] poll() {
    try {
        lock.lock();
        return fsQueue.readNextAndRemove();
    } catch (IOException ex) {
        return null;
    } catch (Exception ex) {
        return null;
    } finally {
        lock.unlock();
    }
}

@Override
public void clear() {
    try {
        lock.lock();
        fsQueue.clear();
    } catch (IOException ex) {
        throw new RuntimeException(ex);
    } catch (Exception e) {
    } finally {
        lock.unlock();
    }
```

```
    }

    /**
     *
     */
    public void close() throws IOException {
        if (fsQueue != null) {
            fsQueue.close();
        }
    }
}

83:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\fqueue\entity\FSQueue.java
 */
package io.nuls.kernel.utils.queue.fqueue.entity;


import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.utils.queue.fqueue.exception.FileEOFException;
import io.nuls.kernel.utils.queue.fqueue.internal.Entity;
import io.nuls.kernel.utils.queue.fqueue.internal.Index;

import java.io.File;
import java.io.IOException;

/**
 * @author opensource
 */
public class FSQueue {
    private long entityLimitLength;
    private String path = null;
    /**
     *
     */
    private Index idx = null;
    private Entity writerHandle = null;
    private Entity readerHandle = null;
    /**
     *
     */
    private int readerIndex = -1;
```

```
private int writerIndex = -1;

public FSQueue(String dir) throws Exception {
    this(new File(dir));
}

/**
 * fileLimitLength
 *
 * @param dir
 * @param entityLimitLength 2G
 */
public FSQueue(String dir, long entityLimitLength) throws Exception {
    this(new File(dir), entityLimitLength);
}

public FSQueue(File dir) throws Exception {
    this(dir, 1024 * 1024 * 2);
}

/**
 * fileLimitLength
 *
 * @param dir
 * @param entityLimitLength 2G
 */
public FSQueue(File dir, long entityLimitLength) throws Exception {
    if (dir.exists() == false && dir.isDirectory() == false) {
        if (dir.mkdirs() == false) {
            throw new IOException("create dir error");
        }
    }
    this.entityLimitLength = entityLimitLength;
    path = dir.getAbsolutePath();
    //
    idx = new Index(path);
    initHandle();
}

private void initHandle() throws Exception {
    writerIndex = idx.getWriterIndex();
    readerIndex = idx.getReaderIndex();
```

```
    writerHandle = new Entity(path, writerIndex, entityLimitLength, idx);
    if (readerIndex == writerIndex) {
        readerHandle = writerHandle;
    } else {
        readerHandle = new Entity(path, readerIndex, entityLimitLength, idx);
    }
}

/**
 * fileLimitLength
 */
private void rotateNextLogWriter() throws Exception {
    writerIndex = (writerIndex + 1) % 1000 + 1;
    writerHandle.putNextFileNumber(writerIndex);
    if (readerHandle != writerHandle) {
        writerHandle.close();
    }
    idx.putWriterIndex(writerIndex);
    writerHandle = new Entity(path, writerIndex, entityLimitLength, idx, true);
}

/**
 * byte
 */
public void add(byte[] message) throws Exception {
    short status = writerHandle.write(message);
    if (status == Entity.WRITEFULL) {
        rotateNextLogWriter();
        status = writerHandle.write(message);
    }
    if (status == Entity.WRITESUCCESS) {
        idx.incrementSize();
    }
}

private byte[] read(boolean commit) throws Exception {
    byte[] bytes = null;
    try {
        bytes = readerHandle.read(commit);
    } catch (FileEOFException e) {
        int nextFileNumber = readerHandle.getNextFileNumber();
        readerHandle.reset();
```

```
            File deleteFile = readerHandle.getFile();
            readerHandle.close();
            deleteFile.delete();
            //
            idx.putReaderPosition(Entity.MESSAGE_START_POSITION);
            idx.putReaderIndex(nextFileNumber);
            if (writerHandle.getCurrentFileNumber() == nextFileNumber) {
                readerHandle = writerHandle;
            } else {
                readerHandle = new Entity(path, nextFileNumber, entityLimitLength, idx);
            }
            try {
                bytes = readerHandle.read(commit);
            } catch (FileEOFException e1) {
                throw new NulsRuntimeException(e1);
            }
        }
        if (bytes != null) {
            idx.decrementSize();
        }
        return bytes;
    }

    /**
     *
     */
    public byte[] readNext() throws Exception {
        return read(false);
    }

    /**
     *
     */
    public byte[] readNextAndRemove() throws Exception {
        return read(true);
    }

    public void clear() throws Exception {
        idx.clear();
        initHandle();
    }
```

```
    public void close() throws IOException {
        readerHandle.close();
        writerHandle.close();
        idx.close();
    }

    public int getQueueSize() {
        return idx.getSize();
    }
}

84:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\fqueue\exception\FileEOFException.java
 */
package io.nuls.kernel.utils.queue.fqueue.exception;

/**
 * @author opensource
 */
public class FileEOFException extends Exception {

    private static final long serialVersionUID = -1L;

    public FileEOFException() {
        super();
    }

    public FileEOFException(String message) {
        super(message);
    }

    public FileEOFException(String message, Throwable cause) {
        super(message, cause);
    }

    public FileEOFException(Throwable cause) {
        super(cause);
    }

    @Override
    public Throwable fillInStackTrace() {
        return this;
```

```
    }

}

85:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\fqueue\internal\Entity.java
 */
package io.nuls.kernel.utils.queue.fqueue.internal;

import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.thread.manager.TaskManager;
import io.nuls.kernel.utils.MappedBufferCleanUtil;
import io.nuls.kernel.utils.queue.fqueue.exception.FileEOFException;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

/**
 *
 *
 * @author opensource
 */
public class Entity {

    public static final byte WRITESUCCESS = 1;
    public static final byte WRITEFAILURE = 2;
    public static final byte WRITEFULL = 3;
    public static final String MAGIC = "FQueuefs";
    public static int MESSAGE_START_POSITION = 20;
    private static final String DB_FILE_PREFIX = "fq_";
    private static final String DB_FILE_SUFFIX = ".db";
    private File file;
    private RandomAccessFile raFile;
    private FileChannel fc;
    private MappedByteBuffer mappedByteBuffer;
    private Index idx = null;
    private long fileLimitLength;
    /**
     *
```

```
 */
private String magicString = null;
private int version = -1;
private int readerPosition = -1;
private int writerPosition = -1;
private int endPosition = -1;
private int currentFileNumber = -1;
private int nextFileNumber = -1;

public Entity(String path, int fileNumber, long fileLimitLength, Index db)
        throws Exception {
    this(path, fileNumber, fileLimitLength, db, false);
}

public Entity(String path, int fileNumber, long fileLimitLength,
              Index idx, boolean create) throws Exception {
    this.currentFileNumber = fileNumber;
    this.fileLimitLength = fileLimitLength;
    this.idx = idx;
    this.file = getIdbFile(path, fileNumber);
    //
    if (!file.exists() || create) {
        createLogEntity();
    } else {
        raFile = new RandomAccessFile(file, "rwd");
        int fileLength = (int) raFile.length();
        if (fileLength < MESSAGE_START_POSITION) {
            throw new Exception("file format error");
        }
        //
        if (fileLimitLength < fileLength) {
            fileLimitLength = fileLength;
            this.fileLimitLength = fileLength;
        }
        fc = raFile.getChannel();
        mappedByteBuffer = fc.map(FileChannel.MapMode.READ_WRITE, 0, fileLimitLength);
        // magicString
        byte[] bytes = new byte[8];
        mappedByteBuffer.get(bytes);
        magicString = new String(bytes);
        if (magicString.equals(MAGIC) == false) {
            throw new Exception("file format error");
```

```
            }
            // version
            version = mappedByteBuffer.getInt();
            // nextFileNumber
            nextFileNumber = mappedByteBuffer.getInt();
            endPosition = mappedByteBuffer.getInt();
            if (endPosition == -1) { //
                writerPosition = idx.getWriterPosition();
            } else { //
                writerPosition = endPosition;
            }
            if (idx.getReaderIndex() == currentFileNumber) {
                readerPosition = idx.getReaderPosition();
            } else {
                readerPosition = MESSAGE_START_POSITION;
            }
        }
        TaskManager.createAndRunThread(NulsConstant.MODULE_ID_MICROKERNEL, path, new
Sync());
    }

    public int getCurrentFileNumber() {
        return currentFileNumber;
    }

    public int getNextFileNumber() {
        return nextFileNumber;
    }

    private boolean createLogEntity() throws IOException {
        raFile = new RandomAccessFile(file, "rwd");
        raFile.setLength(0);
        fc = raFile.getChannel();
        mappedByteBuffer = fc.map(FileChannel.MapMode.READ_WRITE, 0, fileLimitLength);
        mappedByteBuffer.put(MAGIC.getBytes());
        mappedByteBuffer.putInt(version);// 8 version
        mappedByteBuffer.putInt(nextFileNumber);// 12 next fileindex
        mappedByteBuffer.putInt(endPosition);// 16
        magicString = MAGIC;
        writerPosition = MESSAGE_START_POSITION;
        readerPosition = MESSAGE_START_POSITION;
        idx.putWriterPosition(writerPosition);
```

```
        return true;
    }

    public void reset() throws IOException {
        version = -1;
        endPosition = -1;
        currentFileNumber = -1;
        nextFileNumber = -1;
        mappedByteBuffer.position(0);
        mappedByteBuffer.put(MAGIC.getBytes());
        mappedByteBuffer.putInt(version);// 8 version
        mappedByteBuffer.putInt(nextFileNumber);// 12 next fileindex
        mappedByteBuffer.putInt(endPosition);// 16
        mappedByteBuffer.force();
        magicString = MAGIC;
        writerPosition = MESSAGE_START_POSITION;
        readerPosition = MESSAGE_START_POSITION;
    }

    /**
     * write next File number id.
     */
    public void putNextFileNumber(int number) throws IOException {
        mappedByteBuffer.position(12);
        mappedByteBuffer.putInt(number);
        nextFileNumber = number;
    }

    public boolean isFull(int increment) {
        // confirm if the file is full
        if (fileLimitLength < writerPosition + increment) {
            return true;
        }
        return false;
    }

    public byte write(byte[] bytes) throws IOException {
        int increment = bytes.length + 4;
        if (isFull(increment)) {
            mappedByteBuffer.position(16);
            mappedByteBuffer.putInt(writerPosition);
            endPosition = writerPosition;
```

```
            return WRITEFULL;
        }
        mappedByteBuffer.position(writerPosition);
        mappedByteBuffer.putInt(bytes.length);
        mappedByteBuffer.put(bytes);
        writerPosition += increment;
        idx.putWriterPosition(writerPosition);
        return WRITESUCCESS;
    }

    /**
     * @param commit  false
     */
    public byte[] read(boolean commit) throws IOException, FileEOFException {
        if (endPosition != -1 && readerPosition >= endPosition) {
            throw new FileEOFException("file eof");
        }
        if (readerPosition >= writerPosition) {
            return null;
        }
        mappedByteBuffer.position(readerPosition);
        int length = mappedByteBuffer.getInt();
        byte[] bytes = new byte[length];
        mappedByteBuffer.get(bytes);
        if (commit) {
            readerPosition += length + 4;
            idx.putReaderPosition(readerPosition);
        }
        return bytes;
    }

    public File getFile() {
        return file;
    }

    public void close() throws IOException {
        if (mappedByteBuffer != null) {
            mappedByteBuffer.force();
            mappedByteBuffer.clear();
            MappedBufferCleanUtil.clean(mappedByteBuffer);
            mappedByteBuffer = null;
        }
```

```
    if (null != fc) {
        fc.close();
        fc = null;
    }
    if (null != raFile) {
        raFile.close();
        raFile = null;
    }
}

public String headerInfo() {
    StringBuilder sb = new StringBuilder();
    sb.append(" magicString:");
    sb.append(magicString);
    sb.append(" version:");
    sb.append(version);
    sb.append(" readerPosition:");
    sb.append(readerPosition);
    sb.append(" writerPosition:");
    sb.append(writerPosition);
    sb.append(" nextFileNumber:");
    sb.append(nextFileNumber);
    sb.append(" endPosition:");
    sb.append(endPosition);
    sb.append(" currentFileNumber:");
    sb.append(currentFileNumber);
    return sb.toString();
}

public static File getIdbFile(String path, int fileNumber) {
    return new File(path, DB_FILE_PREFIX + fileNumber + DB_FILE_SUFFIX);
}

private class Sync implements Runnable {
    @Override
    public void run() {
        while (mappedByteBuffer != null) {
            try {
                mappedByteBuffer.force();
            } catch (Exception e) {
                break;
            }
```

```
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }
}

86:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\queue\fqueue\internal\Index.java
 */
package io.nuls.kernel.utils.queue.fqueue.internal;


import io.nuls.kernel.utils.MappedBufferCleanUtil;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileChannel.MapMode;
import java.util.concurrent.atomic.AtomicInteger;

/**
 *
 *
 * @author opensource
 */
public class Index {
    private static final int INDEX_LIMIT_LENGTH = 32;
    private static final String INDEX_FILE_NAME = "fq.idx";

    private RandomAccessFile dbRandFile = null;
    private FileChannel fc;
    private MappedByteBuffer mappedByteBuffer;

    /**
     *
```

```
 */
private String magicString = null;
private int version = -1;
private int readerPosition = -1;
private int writerPosition = -1;
private int readerIndex = -1;
private int writerIndex = -1;
private AtomicInteger size = new AtomicInteger();

public Index(String path) throws IOException {
    File dbFile = new File(path, INDEX_FILE_NAME);

    //
    if (dbFile.exists() == false) {
        dbFile.createNewFile();
        dbRandFile = new RandomAccessFile(dbFile, "rwd");
        initIdxFile();
    } else {
        dbRandFile = new RandomAccessFile(dbFile, "rwd");
        if (dbRandFile.length() < INDEX_LIMIT_LENGTH) {
            throw new RuntimeException("file format error.");
        }
        byte[] bytes = new byte[INDEX_LIMIT_LENGTH];
        dbRandFile.read(bytes);
        ByteBuffer buffer = ByteBuffer.wrap(bytes);
        bytes = new byte[Entity.MAGIC.getBytes().length];
        buffer.get(bytes);
        magicString = new String(bytes);
        version = buffer.getInt();
        readerPosition = buffer.getInt();
        writerPosition = buffer.getInt();
        readerIndex = buffer.getInt();
        writerIndex = buffer.getInt();
        int sz = buffer.getInt();
        if (readerPosition == writerPosition && readerIndex == writerIndex && sz <= 0) {
            initIdxFile();
        } else {
            size.set(sz);
        }
    }
    fc = dbRandFile.getChannel();
    mappedByteBuffer = fc.map(MapMode.READ_WRITE, 0, INDEX_LIMIT_LENGTH);
```

```
}

private void initIdxFile() throws IOException {
    magicString = Entity.MAGIC;
    version = 1;
    readerPosition = Entity.MESSAGE_START_POSITION;
    writerPosition = Entity.MESSAGE_START_POSITION;
    readerIndex = 1;
    writerIndex = 1;
    dbRandFile.setLength(32);
    dbRandFile.seek(0);
    dbRandFile.write(magicString.getBytes());// magic
    dbRandFile.writeInt(version);// 8 version
    dbRandFile.writeInt(readerPosition);// 12 reader position
    dbRandFile.writeInt(writerPosition);// 16 write position
    dbRandFile.writeInt(readerIndex);// 20 reader index
    dbRandFile.writeInt(writerIndex);// 24 writer index
    dbRandFile.writeInt(0);// 28 size
}

public void clear() throws IOException {
    mappedByteBuffer.clear();
    mappedByteBuffer.force();
    initIdxFile();
}

/**
 *
 */
public void putWriterPosition(int pos) {
    mappedByteBuffer.position(16);
    mappedByteBuffer.putInt(pos);
    this.writerPosition = pos;
}

/**
 *
 */
public void putReaderPosition(int pos) {
    mappedByteBuffer.position(12);
    mappedByteBuffer.putInt(pos);
    this.readerPosition = pos;
```

```
}

/**
 *
 */
public void putWriterIndex(int index) {
    mappedByteBuffer.position(24);
    mappedByteBuffer.putInt(index);
    this.writerIndex = index;
}

/**
 *
 */
public void putReaderIndex(int index) {
    mappedByteBuffer.position(20);
    mappedByteBuffer.putInt(index);
    this.readerIndex = index;
}

public void incrementSize() {
    int num = size.incrementAndGet();
    mappedByteBuffer.position(28);
    mappedByteBuffer.putInt(num);
}

public void decrementSize() {
    int num = size.decrementAndGet();
    mappedByteBuffer.position(28);
    mappedByteBuffer.putInt(num);
}

public String getMagicString() {
    return magicString;
}

public int getVersion() {
    return version;
}

public int getReaderPosition() {
    return readerPosition;
```

```
}

public int getWriterPosition() {
    return writerPosition;
}

public int getReaderIndex() {
    return readerIndex;
}

public int getWriterIndex() {
    return writerIndex;
}

public int getSize() {
    return size.get();
}

/**
 *
 */
public void close() throws IOException {
    mappedByteBuffer.force();
    mappedByteBuffer.clear();
    MappedBufferCleanUtil.clean(mappedByteBuffer);
    fc.close();
    dbRandFile.close();
    mappedByteBuffer = null;
    fc = null;
    dbRandFile = null;
}

public String headerInfo() {
    StringBuilder sb = new StringBuilder();
    sb.append(" magicString:");
    sb.append(magicString);
    sb.append(" version:");
    sb.append(version);
    sb.append(" readerPosition:");
    sb.append(readerPosition);
    sb.append(" writerPosition:");
    sb.append(writerPosition);
```

```
        sb.append(" size:");
        sb.append(size);
        sb.append(" readerIndex:");
        sb.append(readerIndex);
        sb.append(" writerIndex:");
        sb.append(writerIndex);
        return sb.toString();
    }

}
```

87:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\utils\RestFulUtils.java

```
package io.nuls.kernel.utils;

import io.nuls.core.tools.json.JSONUtils;
import io.nuls.kernel.model.RpcClientResult;
import org.glassfish.jersey.jackson.internal.jackson.jaxrs.json.JacksonJsonProvider;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import java.util.Map;

import static javax.ws.rs.core.MediaType.APPLICATION_JSON;

/**
 * @author Niels
 */
public class RestFulUtils {

    private static RestFulUtils instance = new RestFulUtils();

    private String serverUri;

    private RestFulUtils() {
        client.register(JacksonJsonProvider.class);
    }

    public void setServerUri(String serverUri) {
```

```
        this.serverUri = serverUri;
    }


    public static RestFulUtils getInstance() {
        if (null == instance) {
            throw new RuntimeException("RestFulUtils hasn't inited yet!");
        }
        return instance;
    }

    private Client client = ClientBuilder.newClient();

    public RpcClientResult get(String path, Map<String, Object> params) {
        if (null == serverUri) {
            throw new RuntimeException("service url is null");
        }
        WebTarget target = client.target(serverUri).path(path);
        if (null != params && !params.isEmpty()) {
            for (String key : params.keySet()) {
                target = target.queryParam(key, params.get(key));
            }
        }

        return target.request(APPLICATION_JSON).get(RpcClientResult.class);
    }

    public RpcClientResult post(String path, Map<String, Object> paramsMap) {
        try {
            return post(path, JSONUtils.obj2json(paramsMap));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public RpcClientResult post(String path, String content) {
        if (null == serverUri) {
            throw new RuntimeException("service url is null");
        }
        WebTarget target = client.target(serverUri).path(path);
        return target.request().buildPost(Entity.entity(content,
MediaType.APPLICATION_JSON)).invoke(RpcClientResult.class);
```

```
    }

    public RpcClientResult put(String path, Map<String, Object> paramsMap) {
        try {
            return put(path, JSONUtils.obj2json(paramsMap));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public RpcClientResult put(String path, String content) {
        if (null == serverUri) {
            throw new RuntimeException("service url is null");
        }
        WebTarget target = client.target(serverUri).path(path);
        return target.request().buildPut(Entity.entity(content,
MediaType.APPLICATION_JSON)).invoke(RpcClientResult.class);
    }

    public RpcClientResult delete(String path, Map<String, String> params) {
        if (null == serverUri) {
            throw new RuntimeException("service url is null");
        }
        WebTarget target = client.target(serverUri).path(path);
        if (null != params && !params.isEmpty()) {
            for (String key : params.keySet()) {
                target = target.queryParam(key, params.get(key));
            }
        }
        return target.request().delete(RpcClientResult.class);
    }

}

88:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\SerializeUtils.java
 */
package io.nuls.kernel.utils;

import io.nuls.core.tools.crypto.Sha256Hash;
import io.nuls.core.tools.crypto.Util;
import io.nuls.core.tools.log.Log;
```

```
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.model.NulsData;
import org.spongycastle.crypto.digests.RIPEMD160Digest;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.nio.charset.Charset;
import java.util.List;

/**
 * @author somebody
 */
public class SerializeUtils {

    public static final Charset CHARSET = Charset.forName(NulsConfig.DEFAULT_ENCODING);
    private static final int MAGIC_8 = 8;
    private static final int MAGIC_0X80 = 0x80;
    /**
     * The string that prefixes all text messages signed using Bitcoin keys.
     */
    public static final String SIGNED_MESSAGE_HEADER = "RiceChain Signed Message:\n";
    public static final byte[] SIGNED_MESSAGE_HEADER_BYTES =
SIGNED_MESSAGE_HEADER.getBytes(CHARSET);

    //    /**
//     * MPI encoded numbers are produced by the OpenSSL BN_bn2mpi function. They consist of
//     * a 4 byte big endian length field, followed by the stated number of bytes representing
//     * the number in big endian format (with a sign bit).
//     *
//     * @param hasLength can be set to false if the given array is missing the 4 byte length field
//     */
    public static BigInteger decodeMPI(byte[] mpi, boolean hasLength) {
        byte[] buf;
        if (hasLength) {
            int length = (int) readUint32BE(mpi, 0);
            buf = new byte[length];
            System.arraycopy(mpi, 4, buf, 0, length);
```

```
        } else {
            buf = mpi;
        }
        if (buf.length == 0) {
            return BigInteger.ZERO;
        }
        boolean isNegative = (buf[0] & MAGIC_0X80) == MAGIC_0X80;
        if (isNegative) {
            buf[0] &= 0x7f;
        }
        BigInteger result = new BigInteger(buf);
        return isNegative ? result.negate() : result;
    }

    //    /**
//     * MPI encoded numbers are produced by the OpenSSL BN_bn2mpi function. They consist of
//     * a 4 byte big endian length field, followed by the stated number of bytes representing
//     * the number in big endian format (with a sign bit).
//     *
//     * @param includeLength indicates whether the 4 byte length field should be included
//     */
    public static byte[] encodeMPI(BigInteger value, boolean includeLength) {
        if (value.equals(BigInteger.ZERO)) {
            if (!includeLength) {
                return new byte[]{};
            } else {
                return new byte[]{0x00, 0x00, 0x00, 0x00};
            }
        }
        boolean isNegative = value.signum() < 0;
        if (isNegative) {
            value = value.negate();
        }
        byte[] array = value.toByteArray();
        int length = array.length;
        if ((array[0] & MAGIC_0X80) == MAGIC_0X80) {
            length++;
        }
        if (includeLength) {
            byte[] result = new byte[length + 4];
            System.arraycopy(array, 0, result, length - array.length + 3, array.length);
            uint32ToByteArrayBE(length, result, 0);
```

```
            if (isNegative) {
                result[4] |= MAGIC_0X80;
            }
            return result;
        } else {
            byte[] result;
            if (length != array.length) {
                result = new byte[length];
                System.arraycopy(array, 0, result, 1, array.length);
            } else {
                result = array;
            }
            if (isNegative) {
                result[0] |= MAGIC_0X80;
            }
            return result;
        }
    }

    //
//    /**
//     * Given a textual message, returns a byte buffer formatted as follows:</p>
//     * <p>
//     * <tt>[24] "Bitcoin Signed Message:\n" [message.length as a varint] message</p></tt>
//     */
    public static byte[] formatMessageForSigning(String message) {
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            bos.write(SIGNED_MESSAGE_HEADER_BYTES.length);
            bos.write(SIGNED_MESSAGE_HEADER_BYTES);
            byte[] messageBytes = message.getBytes(CHARSET);
            VarInt size = new VarInt(messageBytes.length);
            bos.write(size.encode());
            bos.write(messageBytes);
            return bos.toByteArray();
        } catch (IOException e) {
            // Cannot happen.
            throw new RuntimeException(e);
        }
    }

    public static String toString(byte[] bytes, String charsetName) {
```

```
        try {
            return new String(bytes, charsetName);
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }

    public static byte[] toBytes(CharSequence str, String charsetName) {
        try {
            return str.toString().getBytes(charsetName);
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException(e);
        }
    }

    //    /**
//     * Returns a copy of the given byte array in reverse order.
//     */
    public static byte[] reverseBytes(byte[] bytes) {
        return Util.reverseBytes(bytes);
    }

    //    /**
//     * Parse 4 bytes from the byte array (starting at the offset) as unsigned 32-bit integer in little
endian format.
//     */
    public static long readUint32LE(byte[] bytes, int offset) {
        return (bytes[offset] & 0xffL) |
                ((bytes[offset + 1] & 0xffL) << 8) |
                ((bytes[offset + 2] & 0xffL) << 16) |
                ((bytes[offset + 3] & 0xffL) << 24);
    }

    public static int readUint16LE(byte[] bytes, int offset) {
        return (bytes[offset] & 0xff) |
                ((bytes[offset + 1] & 0xff) << 8);
    }

    public static int readInt32LE(byte[] bytes, int offset) {
        return (bytes[offset] & 0xff) |
                ((bytes[offset + 1] & 0xff) << 8) |
                ((bytes[offset + 2] & 0xff) << 16) |
```

```
                ((bytes[offset + 3] & 0xff) << 24);
    }

    //    /**
//     * Parse 8 bytes from the byte array (starting at the offset) as signed 64-bit integer in little endian format.
//     */
    public static long readInt64LE(byte[] bytes, int offset) {
        return (bytes[offset] & 0xffL) |
                ((bytes[offset + 1] & 0xffL) << 8) |
                ((bytes[offset + 2] & 0xffL) << 16) |
                ((bytes[offset + 3] & 0xffL) << 24) |
                ((bytes[offset + 4] & 0xffL) << 32) |
                ((bytes[offset + 5] & 0xffL) << 40) |
                ((bytes[offset + 6] & 0xffL) << 48) |
                ((bytes[offset + 7] & 0xffL) << 56);
    }

    //    /**
//     * Parse 4 bytes from the byte array (starting at the offset) as unsigned 32-bit integer in big endian format.
//     */
    public static long readUint32BE(byte[] bytes, int offset) {
        return ((bytes[offset] & 0xffL) << 24) |
                ((bytes[offset + 1] & 0xffL) << 16) |
                ((bytes[offset + 2] & 0xffL) << 8) |
                (bytes[offset + 3] & 0xffL);
    }

    //    /**
//     * Parse 2 bytes from the byte array (starting at the offset) as unsigned 16-bit integer in big endian format.
//     */
    public static int readUint16BE(byte[] bytes, int offset) {
        return ((bytes[offset] & 0xff) << 8) |
                (bytes[offset + 1] & 0xff);
    }

    //
//    /**
//     * Calculates RIPEMD160(SHA256(input)). This is used in Address calculations.
//     */
```

```
    public static byte[] sha256hash160(byte[] input) {
        byte[] sha256 = Sha256Hash.hash(input);
        RIPEMD160Digest digest = new RIPEMD160Digest();
        digest.update(sha256, 0, sha256.length);
        byte[] out = new byte[20];
        digest.doFinal(out, 0);
        return out;
    }

    //    /**
//     * The regular {@link 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 void uint32ToByteArrayBE(long val, byte[] out, int offset) {
        out[offset] = (byte) (0xFF & (val >> 24));
        out[offset + 1] = (byte) (0xFF & (val >> 16));
        out[offset + 2] = (byte) (0xFF & (val >> 8));
        out[offset + 3] = (byte) (0xFF & val);
    }

    //    /** Write 2 bytes to the output stream as unsigned 16-bit integer in little endian format. */
    public static void uint16ToByteStreamLE(int val, OutputStream stream) throws IOException {
        stream.write((int) (0xFF & val));
        stream.write((int) (0xFF & (val >> 8)));
    }
```

```
public static void int16ToByteArrayLE(short val, byte[] out, int offset) {
    out[offset] = (byte) (0xFF & val);
    out[offset + 1] = (byte) (0xFF & (val >> 8));
}

public static void uint32ToByteArrayLE(long val, byte[] out, int offset) {
    out[offset] = (byte) (0xFF & val);
    out[offset + 1] = (byte) (0xFF & (val >> 8));
    out[offset + 2] = (byte) (0xFF & (val >> 16));
    out[offset + 3] = (byte) (0xFF & (val >> 24));
}

public static void int32ToByteArrayLE(int val, byte[] out, int offset) {
    out[offset] = (byte) (0xFF & val);
    out[offset + 1] = (byte) (0xFF & (val >> 8));
    out[offset + 2] = (byte) (0xFF & (val >> 16));
    out[offset + 3] = (byte) (0xFF & (val >> 24));
}

public static void uint64ToByteArrayLE(long val, byte[] out, int offset) {
    out[offset] = (byte) (0xFF & val);
    out[offset + 1] = (byte) (0xFF & (val >> 8));
    out[offset + 2] = (byte) (0xFF & (val >> 16));
    out[offset + 3] = (byte) (0xFF & (val >> 24));
    out[offset + 4] = (byte) (0xFF & (val >> 32));
    out[offset + 5] = (byte) (0xFF & (val >> 40));
    out[offset + 6] = (byte) (0xFF & (val >> 48));
    out[offset + 7] = (byte) (0xFF & (val >> 56));
}

public static void int16ToByteStreamLE(short val, OutputStream stream) throws IOException {
    stream.write((byte) (0xFF & val));
    stream.write((byte) (0xFF & (val >> 8)));
}

public static void uint32ToByteStreamLE(long val, OutputStream stream) throws IOException {
    stream.write((int) (0xFF & val));
    stream.write((int) (0xFF & (val >> 8)));
    stream.write((int) (0xFF & (val >> 16)));
    stream.write((int) (0xFF & (val >> 24)));
}
```

```
    public static void int64ToByteStreamLE(long val, OutputStream stream) throws IOException {
        stream.write((int) (0xFF & val));
        stream.write((int) (0xFF & (val >> 8)));
        stream.write((int) (0xFF & (val >> 16)));
        stream.write((int) (0xFF & (val >> 24)));
        stream.write((int) (0xFF & (val >> 32)));
        stream.write((int) (0xFF & (val >> 40)));
        stream.write((int) (0xFF & (val >> 48)));
        stream.write((int) (0xFF & (val >> 56)));
    }

    public static void uint64ToByteStreamLE(BigInteger val, OutputStream stream) throws
IOException {
        byte[] bytes = val.toByteArray();
        if (bytes.length > MAGIC_8) {
            throw new RuntimeException("Input too large to encode into a uint64");
        }
        bytes = reverseBytes(bytes);
        stream.write(bytes);
        if (bytes.length < MAGIC_8) {
            for (int i = 0; i < MAGIC_8 - bytes.length; i++) {
                stream.write(0);
            }
        }
    }

    public static void doubleToByteStream(double val, OutputStream stream) throws IOException {
        stream.write(double2Bytes(val));
    }

    //    /**
//     * doublebyte
//     *
//     * @return byte[]
//     */
    public static byte[] double2Bytes(double d) {
        long value = Double.doubleToRawLongBits(d);
        byte[] byteRet = new byte[MAGIC_8];
        for (int i = 0; i < MAGIC_8; i++) {
            byteRet[i] = (byte) ((value >> 8 * i) & 0xff);
        }
```

```
        return byteRet;
    }

    //    /**
//     * byte[]double
//     *
//     * @return double
//     */
    public static double bytes2Double(byte[] arr) {
        long value = 0;
        for (int i = 0; i < MAGIC_8; i++) {
            value |= ((long) (arr[i] & 0xff)) << (MAGIC_8 * i);
        }
        return Double.longBitsToDouble(value);
    }

    public static String join(List<? extends Object> list) {
        if (list == null) {
            return null;
        }
        StringBuilder sb = new StringBuilder();
        for (Object object : list) {
            if (sb.length() != 0) {
                sb.append(" ");
            }
            sb.append(object.toString());
        }
        return sb.toString();
    }

    public static short bytes2Short(byte[] b) {
        return (short) (((b[1] << 8) | b[0] & 0xff));
    }

    public static byte[] shortToBytes(short val) {
        byte[] bytes = new byte[2];
        bytes[1] = (byte) (0xFF & val >> 8);
        bytes[0] = (byte) (0xFF & val >> 0);
        return bytes;
    }

    public static byte[] int32ToBytes(int x) {
```

```
    byte[] bb = new byte[4];
    bb[3] = (byte) (0xFF & x >> 24);
    bb[2] = (byte) (0xFF & x >> 16);
    bb[1] = (byte) (0xFF & x >> 8);
    bb[0] = (byte) (0xFF & x >> 0);
    return bb;
}

public static byte[] int16ToBytes(int x) {
    byte[] bb = new byte[2];
    bb[1] = (byte) (0xFF & x >> 8);
    bb[0] = (byte) (0xFF & x >> 0);
    return bb;
}

public static long randomLong() {
    return (long) (Math.random() * Long.MAX_VALUE);
}

public static int sizeOfDouble(Double val) {
    return MAGIC_8;
}

public static int sizeOfString(String val) {
    if (null == val) {
        return 1;
    }
    byte[] bytes;
    try {
        bytes = val.getBytes(NulsConfig.DEFAULT_ENCODING);
    } catch (UnsupportedEncodingException e) {
        Log.error(e);
        throw new NulsRuntimeException(e);
    }
    return sizeOfBytes(bytes);
}

public static int sizeOfVarInt(Long val) {
    return VarInt.sizeOf(val);
}

public static int sizeOfUint48() {
```

```
    return NulsConstant.INT48_VALUE_LENGTH;
}

public static int sizeOfInt32() {
    return 4;
}

public static int sizeOfUint32() {
    return 4;
}

public static int sizeOfInt16() {
    return 2;
}

public static int sizeOfUint16() {
    return 2;
}

public static int sizeOfInt64() {
    return 8;
}

public static int sizeOfVarInt(Integer val) {
    return VarInt.sizeOf(val);
}

public static int sizeOfBoolean(Boolean val) {
    return 1;
}

public static int sizeOfBytes(byte[] val) {
    if (null == val) {
        return 1;
    }
    return VarInt.sizeOf((val).length) + (val).length;
}

public static int sizeOfNulsData(NulsData val) {
    if (null == val) {
        return NulsConstant.PLACE_HOLDER.length;
    }
```

```
        int size = val.size();
        return size == 0 ? 1 : size;
    }

    public static byte[] uint64ToByteArray(long val) {
        byte[] out = new byte[8];
        out[0] = (byte) (0xFF & val);
        out[1] = (byte) (0xFF & (val >> 8));
        out[2] = (byte) (0xFF & (val >> 16));
        out[3] = (byte) (0xFF & (val >> 24));
        out[4] = (byte) (0xFF & (val >> 32));
        out[5] = (byte) (0xFF & (val >> 40));
        out[6] = (byte) (0xFF & (val >> 48));
        out[7] = (byte) (0xFF & (val >> 56));
        return out;
    }

    public static byte[] uint48ToBytes(long val) {
        byte[] bytes = new byte[SerializeUtils.sizeOfUint48()];
        bytes[0] = (byte) (0xFF & val);
        bytes[1] = (byte) (0xFF & (val >> 8));
        bytes[2] = (byte) (0xFF & (val >> 16));
        bytes[3] = (byte) (0xFF & (val >> 24));
        bytes[4] = (byte) (0xFF & (val >> 32));
        bytes[5] = (byte) (0xFF & (val >> 40));
        return bytes;
    }

    //    /** Parse 2 bytes from the byte array (starting at the offset) as unsigned 16-bit integer in little endian format. */
    public static int readUint16(byte[] bytes, int offset) {
        return (bytes[offset] & 0xff) |
                ((bytes[offset + 1] & 0xff) << 8);
    }

    //    /** Parse 4 bytes from the byte array (starting at the offset) as unsigned 32-bit integer in little endian format. */
    public static long readUint32(byte[] bytes, int offset) {
        return (bytes[offset] & 0xffL) |
                ((bytes[offset + 1] & 0xffL) << 8) |
                ((bytes[offset + 2] & 0xffL) << 16) |
                ((bytes[offset + 3] & 0xffL) << 24);
```

```
    }

    public static long readInt64(byte[] bytes, int offset) {
        return (bytes[offset] & 0xffL) |
                ((bytes[offset + 1] & 0xffL) << 8) |
                ((bytes[offset + 2] & 0xffL) << 16) |
                ((bytes[offset + 3] & 0xffL) << 24) |
                ((bytes[offset + 4] & 0xffL) << 32) |
                ((bytes[offset + 5] & 0xffL) << 40) |
                ((bytes[offset + 6] & 0xffL) << 48) |
                ((bytes[offset + 7] & 0xffL) << 56);
    }

}

89:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\utils\TransactionFeeCalculator.java
 */

package io.nuls.kernel.utils;

import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.model.Na;

/**
 * @author Niels
 */
public class TransactionFeeCalculator {

    public static final Na MIN_PRECE_PRE_1024_BYTES = Na.valueOf(100000);
    public static final Na OTHER_PRECE_PRE_1024_BYTES = Na.valueOf(1000000);

    public static final int KB = 1024;

    //    /**
//     *
//     * According to the transaction size calculate the handling fee.
//     *
//     * @param size /size of the transaction
//     */
    public static final Na getTransferFee(int size) {
```

```
        Na fee = MIN_PRECE_PRE_1024_BYTES.multiply(size / KB);
        if (size % KB > 0) {
            fee = fee.add(MIN_PRECE_PRE_1024_BYTES);
        }
        return fee;
    }

    //    /**
//     *
//     * According to the transaction size calculate the handling fee.
//     *
//     * @param size /size of the transaction
//     */
    public static final Na getMaxFee(int size) {
        Na fee = OTHER_PRECE_PRE_1024_BYTES.multiply(size / KB);
        if (size % KB > 0) {
            fee = fee.add(OTHER_PRECE_PRE_1024_BYTES);
        }
        return fee;
    }

    //    /**
//     *
//     * According to the transaction size calculate the handling fee.
//     *
//     * @param size /size of the transaction
//     */
    public static final Na getFee(int size, Na price) {
        if (price.isLessThan(MIN_PRECE_PRE_1024_BYTES)) {
            throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
        }
        if (price.isGreaterThan(OTHER_PRECE_PRE_1024_BYTES)) {
            throw new NulsRuntimeException(KernelErrorCode.DATA_ERROR);
        }
        Na fee = price.multiply(size / KB);
        if (size % KB > 0) {
            fee = fee.add(price);
        }
        return fee;
    }
}
```

```
90:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\utils\TransactionManager.java
 */
package io.nuls.kernel.utils;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.processor.TransactionProcessor;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @author Niels
 */
public class TransactionManager {

    private static final Map<Class<? extends Transaction>, Class<? extends TransactionProcessor>> TX_SERVICE_MAP = new HashMap<>();
    private static final Map<Integer, Class<? extends Transaction>> TYPE_TX_MAP = new HashMap<>();

    public static void init() throws Exception {
        List<TransactionProcessor> beanList = SpringLiteContext.getBeanList(TransactionProcessor.class);
        for (TransactionProcessor processor : beanList) {
            registerProcessor(processor);
        }
    }

    private static void registerProcessor(TransactionProcessor processor) {
        Method[] methods = processor.getClass().getDeclaredMethods();
        for (Method method : methods) {
            if (!"onCommit".equals(method.getName())) {
                continue;
            }
```

```
            Class paramType = method.getParameterTypes()[0];
            if (paramType.equals(Transaction.class)) {
                continue;
            }
            putTx(paramType, processor.getClass());
            break;
        }
    }

    public static final void putTx(Class<? extends Transaction> txClass, Class<? extends
TransactionProcessor> txProcessorClass) {
        if (null != txProcessorClass) {
            TX_SERVICE_MAP.put(txClass, txProcessorClass);
        }
        try {
            Transaction tx = txClass.newInstance();
            TYPE_TX_MAP.put(tx.getType(), txClass);
        } catch (InstantiationException e) {
            Log.error(e);
        } catch (IllegalAccessException e) {
            Log.error(e);
        }
    }

    private static TransactionProcessor getProcessor(Class<? extends Transaction> txClass) {
        Class<? extends TransactionProcessor> txProcessorClass =
TX_SERVICE_MAP.get(txClass);
        if (null == txProcessorClass) {
            return null;
        }
        return SpringLiteContext.getBean(txProcessorClass);
    }

    public static List<TransactionProcessor> getProcessorList(Class<? extends Transaction>
txClass) {
        List<TransactionProcessor> list = new ArrayList<>();
        Class clazz = txClass;
        while (!clazz.equals(Transaction.class)) {
            TransactionProcessor txService = TransactionManager.getProcessor(clazz);
            if (null != txService) {
                list.add(0, txService);
            }
```

```
            clazz = clazz.getSuperclass();
        }
        return list;
    }

    public static List<TransactionProcessor> getAllProcessorList() {
        try {
            return SpringLiteContext.getBeanList(TransactionProcessor.class);
        } catch (Exception e) {
            Log.error(e);
            return new ArrayList<>();
        }
    }

    public static Transaction getInstance(NulsByteBuffer byteBuffer) throws Exception {
        int txType = byteBuffer.readUint16();
        byteBuffer.setCursor(byteBuffer.getCursor() - SerializeUtils.sizeOfUint16());
        Class<? extends Transaction> txClass = TYPE_TX_MAP.get(txType);
        if (null == txClass) {
            throw new NulsRuntimeException(KernelErrorCode.DATA_NOT_FOUND);
        }
        Transaction tx = byteBuffer.readNulsData(txClass.newInstance());
        return tx;
    }


    public static List<Transaction> getInstances(NulsByteBuffer byteBuffer, long txCount) throws
Exception {
        List<Transaction> list = new ArrayList<>();
        for (int i = 0; i < txCount; i++) {
            list.add(getInstance(byteBuffer));
        }
        return list;
    }
}

91:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\utils\VarInt.java
 */


package io.nuls.kernel.utils;
```

```
/**
 * A variable-length encoded unsigned integer using Satoshi's encoding (a.k.a. "CompactSize").
 */
public class VarInt {
    public final long value;
    private final int originallyEncodedSize;

    /**
     * Constructs a new VarInt with the given unsigned long value.
     *
     * @param value the unsigned long value (beware widening conversion of negatives!)
     */
    public VarInt(long value) {
        this.value = value;
        originallyEncodedSize = getSizeInBytes();
    }

    /**
     * Constructs a new VarInt with the value parsed from the specified offset of the given buffer.
     *
     * @param buf    the buffer containing the value
     * @param offset the offset of the value
     */
    public VarInt(byte[] buf, int offset) {
        int first = 0xFF & buf[offset];
        if (first < 253) {
            value = first;
            // 1 data byte (8 bits)
            originallyEncodedSize = 1;
        } else if (first == 253) {
            value = (0xFF & buf[offset + 1]) | ((0xFF & buf[offset + 2]) << 8);
            // 1 marker + 2 data bytes (16 bits)
            originallyEncodedSize = 3;
        } else if (first == 254) {
            value = SerializeUtils.readUint32LE(buf, offset + 1);
            // 1 marker + 4 data bytes (32 bits)
            originallyEncodedSize = 5;
        } else {
            value = SerializeUtils.readInt64LE(buf, offset + 1);
            // 1 marker + 8 data bytes (64 bits)
            originallyEncodedSize = 9;
```

```
        }
    }

    //    /**
//     * Returns the original number of bytes used to encode the value if it was
//     * deserialized from a byte array, or the minimum encoded size if it was not.
//     */
    public int getOriginalSizeInBytes() {
        return originallyEncodedSize;
    }

    //    /**
//     * Returns the minimum encoded size of the value.
//     */
    public final int getSizeInBytes() {
        return sizeOf(value);
    }

    //    /**
//     * Returns the minimum encoded size of the given unsigned long value.
//     *
//     * @param value the unsigned long value (beware widening conversion of negatives!)
//     */
    public static int sizeOf(long value) {
        // if negative, it's actually a very large unsigned long value
        if (value < 0) {
            // 1 marker + 8 data bytes
            return 9;
        }
        if (value < 253) {
            // 1 data byte
            return 1;
        }
        if (value <= 0xFFFFL) {
            // 1 marker + 2 data bytes
            return 3;
        }
        if (value <= 0xFFFFFFFFL) {
            // 1 marker + 4 data bytes
            return 5;
        }
        // 1 marker + 8 data bytes
```

```
        return 9;
    }

    //    /**
//     * Encodes the value into its minimal representation.
//     *
//     * @return the minimal encoded bytes of the value
//     */
    public byte[] encode() {
        byte[] bytes;
        switch (sizeOf(value)) {
            case 1:
                return new byte[]{(byte) value};
            case 3:
                return new byte[]{(byte) 253, (byte) (value), (byte) (value >> 8)};
            case 5:
                bytes = new byte[5];
                bytes[0] = (byte) 254;
                SerializeUtils.uint32ToByteArrayLE(value, bytes, 1);
                return bytes;
            default:
                bytes = new byte[9];
                bytes[0] = (byte) 255;
                SerializeUtils.uint64ToByteArrayLE(value, bytes, 1);
                return bytes;
        }
    }
}

92:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\validate\DataValidatorChain.java
 */
package io.nuls.kernel.validate;

import io.nuls.core.tools.log.Log;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.NulsData;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
```

```
import java.util.Set;

/**
 * @author Niels
 */
public class DataValidatorChain {

    private List<NulsDataValidator<NulsData>> list = new ArrayList<>();
    private Set<Class> classSet = new HashSet<>();
    private ThreadLocal<Integer> index = new ThreadLocal<>();

    public ValidateResult startDoValidator(NulsData data) {
        if (list.isEmpty()) {
            return ValidateResult.getSuccessResult();
        }
        index.set(-1);
        ValidateResult result;
        try {
            result = doValidate(data);
        } catch (Exception e) {
            Log.error(e);
            result = ValidateResult.getFailedResult(this.getClass().getName(),
KernelErrorCode.SYS_UNKOWN_EXCEPTION);
        }
        boolean b = index.get() == list.size();
        index.remove();
        if (!b && result.isSuccess()) {
            return ValidateResult.getFailedResult(this.getClass().getName(),
KernelErrorCode.VALIDATORS_NOT_FULLY_EXECUTED);
        }
        return result;
    }

    private ValidateResult doValidate(NulsData data) {
        index.set(1 + index.get());
        if (index.get() == list.size()) {
            return ValidateResult.getSuccessResult();
        }
        NulsDataValidator validator = list.get(index.get());
        ValidateResult result = null;
        try {
            result = validator.validate(data);
```

```
        } catch (NulsException e) {
            Log.error(e);
            return ValidateResult.getFailedResult(this.getClass().getName(), e.getErrorCode());
        }
        if (null == result) {
            Log.error(validator.getClass() + " has null result!");
        }
        if (!result.isSuccess()) {
            return result;
        }
        return this.doValidate(data);
    }

    public void addValidator(NulsDataValidator validator) {
        if (null == validator) {
            return;
        }

        if (classSet.add(validator.getClass())) {
            list.add(validator);
        }
    }
}
```

93:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\validate\NulsDataValidator.java

```
 */
package io.nuls.kernel.validate;

import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.NulsData;

/**
 * @author Niels
 */
public interface NulsDataValidator<T extends NulsData> {

    ValidateResult validate(T data) throws NulsException;
}
```

94:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java\io\nuls\kernel\validate\ValidateResult.java

```
 */
package io.nuls.kernel.validate;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.kernel.constant.ErrorCode;
import io.nuls.kernel.constant.SeverityLevelEnum;
import io.nuls.kernel.model.Result;

/**
 * @author Niels
 */
public class ValidateResult<T> extends Result<T> {

    private SeverityLevelEnum level;

    private String className;

    public static ValidateResult getSuccessResult() {
        ValidateResult result = new ValidateResult();
        result.setSuccess(true);
        result.setMsg("");
        return result;
    }

    public static ValidateResult getFailedResult(String className, ErrorCode msg) {
        return getFailedResult(className, SeverityLevelEnum.WRONG, msg);
    }

    public static ValidateResult getFailedResult(String className, SeverityLevelEnum level,
ErrorCode errorCode) {
        ValidateResult result = new ValidateResult();
        result.setSuccess(false);
        result.setLevel(level);
        result.setErrorCode(errorCode);
        result.setClassName(className);
        return result;
    }

    @JsonIgnore
    public SeverityLevelEnum getLevel() {
        return level;
    }
```

```
    public void setLevel(SeverityLevelEnum level) {
        this.level = level;
    }


    @JsonIgnore
    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }
}

95:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\main\java\io\nuls\kernel\validate\ValidatorManager.java
 */

package io.nuls.kernel.validate;

import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.exception.NulsRuntimeException;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.model.BaseNulsData;
import io.nuls.kernel.model.NulsData;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author Niels
 */
public class ValidatorManager {

    private static Map<Class, DataValidatorChain> chainMap = new ConcurrentHashMap<>();

    private static boolean success;
```

```
    public static void init() {
        List<NulsDataValidator> validatorList = null;
        try {
            validatorList = SpringLiteContext.getBeanList(NulsDataValidator.class);
        } catch (Exception e) {
            throw new NulsRuntimeException(e);
        }
        for (NulsDataValidator validator : validatorList) {
            Method[] methods = validator.getClass().getDeclaredMethods();
            for (Method method : methods) {
                if ("validate".equals(method.getName())) {
                    Class paramType = method.getParameterTypes()[0];
                    if (paramType.equals(NulsData.class)) {
                        continue;
                    }
                    addValidator(paramType, validator);
                    break;
                }
            }
        }
        success = true;
    }

    public static boolean isInitSuccess() {
        return success;
    }

    public static void addValidator(Class<? extends NulsData> clazz, NulsDataValidator<? extends
NulsData> validator) {
        DataValidatorChain chain = chainMap.get(clazz);
        if (null == chain) {
            chain = new DataValidatorChain();
            chainMap.put(clazz, chain);
        }
        chain.addValidator(validator);
    }

    public static ValidateResult startDoValidator(NulsData data) {
        if (data == null) {
            return ValidateResult.getFailedResult(ValidatorManager.class.getName(),
KernelErrorCode.NULL_PARAMETER);
```

```
        }
        List<DataValidatorChain> chainList = new ArrayList<>();
        Class<NulsData> clazz = (Class<NulsData>) data.getClass();
        while (!clazz.equals(BaseNulsData.class)) {
            DataValidatorChain chain = chainMap.get(clazz);
            if (null != chain) {
                chainList.add(chain);
            }
            clazz = (Class<NulsData>) clazz.getSuperclass();
        }
        for (DataValidatorChain chain : chainList) {
            ValidateResult result = chain.startDoValidator(data);
            if (result.isFailed()) {
                return result;
            }
        }
        return ValidateResult.getSuccessResult();
    }

}


96:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\main\java-
templates\io\nuls\module\version\KernelMavenInfo.java
 */

package io.nuls.module.version;

import io.nuls.kernel.model.NulsVersion;
import io.nuls.kernel.lite.annotation.Component;

/**
 * @author: Niels Wang
 */
@Component
public class KernelMavenInfo implements NulsVersion {

    public static final String VERSION = "${project.version}";
    public static final String GROUP_ID = "${project.groupId}";
    public static final String ARTIFACT_ID = "${project.artifactId}";

    public String getVersion() {
```

```
        return VERSION;
    }

    public String getArtifactId() {
        return ARTIFACT_ID;
    }

    public String getGroupId() {
        return GROUP_ID;
    }


}
```

97:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\test\java\io\nuls\kernel\func\TimeServiceTest.java

```
 */
package io.nuls.kernel.func;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * @author Niels
 * @date 2018/7/7
 */
public class TimeServiceTest {

    public static void main(String[] args) {
        TimeService timeService = TimeService.getInstance();

        timeService.run();

        assertNotEquals(0L,TimeService.getNetTimeOffset());
    }
}
```

98:F:\git\coin\nuls\nuls-1.1.3\nuls\core-module\kernel\src\test\java\io\nuls\kernel\MicroKernelBootstrapTest.java

```
 */
```

```
package io.nuls.kernel;

import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.lite.core.SpringLiteContext;
import io.nuls.kernel.validate.ValidatorManager;
import org.junit.Test;

import static org.junit.Assert.*;

/**
 * @author: Niels Wang
 */
public class MicroKernelBootstrapTest {

    private MicroKernelBootstrap bootstrap = MicroKernelBootstrap.getInstance();

    @Test
    public void init() {
        bootstrap.init();
        assertNotNull(NulsConfig.MODULES_CONFIG);
        assertNotNull(NulsConfig.NULS_CONFIG);
        assertTrue(SpringLiteContext.isInitSuccess());
        assertNotNull(NulsConfig.VERSION);
        assertNotNull(ValidatorManager.isInitSuccess());
    }

    @Test
    public void test(){
        getInfo();
        start();
        shutdown();
        destroy();
    }

    public void getInfo() {
        this.init();
        String info = bootstrap.getInfo();
        System.out.println(info);
        assertTrue(true);
    }

    public void start() {
```

```
        bootstrap.start();
        assertTrue(true);
    }

    public void shutdown() {
        bootstrap.shutdown();
        assertTrue(true);
    }

    public void destroy() {
        bootstrap.destroy();
        assertTrue(true);
    }

}

99:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\test\java\io\nuls\kernel\model\NulsDigestDataTest.java
 */

package io.nuls.kernel.model;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * @author: Niels Wang
 */
public class NulsDigestDataTest {

    @Test
    public void test() {
        NulsDigestData hash = NulsDigestData.calcDigestData(new byte[32]);
        System.out.println(hash);
        assertTrue(true);
    }
}

100:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\test\java\io\nuls\kernel\type\Int48Test.java
 */
```

```
package io.nuls.kernel.type;

import io.nuls.kernel.utils.SerializeUtils;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

/**
 * @author Niels
 */
public class Int48Test {
    @Test
    public void test() {
        long time = -1L;
        byte[] bytes = SerializeUtils.uint48ToBytes(time);
        long value = readInt48(bytes);
//        assertEquals(value, time);
    }

    @Test
    public void testLong() {
        byte[] bytes = new byte[]{-1,-1,-1,-1,-1,-1,0,0};
        long value = SerializeUtils.readInt64LE(bytes,0);
        System.out.println(value);
    }

    private long readInt48(byte[] bytes) {
        long value = (bytes[0] & 0xffL) |
                ((bytes[1] & 0xffL) << 8) |
                ((bytes[2] & 0xffL) << 16) |
                ((bytes[3] & 0xffL) << 24) |
                ((bytes[4] & 0xffL) << 32) |
                ((bytes[5] & 0xffL) << 40) ;
        return value;
    }
}

101:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\test\java\io\nuls\kernel\utils\queue\entity\PersistentQueueTest.java
 */
```

```
package io.nuls.kernel.utils.queue.entity;

import io.nuls.kernel.model.Transaction;
import org.junit.Test;

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

import static org.junit.Assert.*;

/**
 * @author: Niels Wang
 * @date: 2018/7/8
 */
public class PersistentQueueTest {

    private List<Transaction> txList;

    @Test
    public void test() throws Exception {
//        FQueue  txQueue = new FQueue("tx-1", 1000001L);
        PersistentQueue txQueue = new PersistentQueue("tx-121", 1000001L);
        this.txList = new ArrayList<>();
        for (int i = 0; i < 1000000; i++) {
            Transaction tx = new TestTransaction();
            tx.setTime(i);
tx.setRemark("sdfsdfsdfsdfsdfsdfaaadsfasdfsadfsdfasdfasdfasdfasdfasdfsadfaaaaaaaaaaaaaaaaaa
aaaaabsdsadfsadfsdfsdfsdfsdfsdfsdfsdfaaadsfasdfsadfsdfasdfasdfasdfasdfasdfsadfaaaaaaaaaaa
aaaaaaaaaaabsdsadfsadfsdfsdfsdfsdfsdfsdfsdfaa".getBytes());
            txList.add(tx);
        }

        long start = System.currentTimeMillis();
        for (Transaction tx : txList) {
            txQueue.offer(tx.serialize());
        }
        System.out.println("100" + (System.currentTimeMillis() - start) + "ms");

        start = System.currentTimeMillis();
        int i = 0;
        while (true) {
            i++;
```

```
            byte[] bytes = txQueue.poll();
            System.out.println(i + "::::::" + bytes);
            if (null == bytes) {
                break;
            }
        }
        System.out.println("100" + (System.currentTimeMillis() - start) + "ms");
//
        assertTrue(true);
    }
}

102:F:\git\coin\nuls\nuls-1.1.3\nuls\core-
module\kernel\src\test\java\io\nuls\kernel\utils\queue\entity\TestTransaction.java
 */

package io.nuls.kernel.utils.queue.entity;

import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;

/**
 * @author: Niels Wang
 * @date: 2018/7/8
 */
public class TestTransaction extends Transaction {
    public TestTransaction( ) {
        super(101);
    }

    @Override
    protected TransactionLogicData parseTxData(NulsByteBuffer byteBuffer) throws NulsException
{
        return null;
    }

    @Override
    public String getInfo(byte[] address) {
        return null;
    }
```

```
}

103:F:\git\coin\nuls\nuls-1.1.3\nuls\db-
module\db\src\main\java\io\nuls\db\constant\DBConstant.java
public interface DBConstant extends NulsConstant {

    short MODULE_ID_DB = 2;

    String BASE_AREA_NAME = "base";

}

104:F:\git\coin\nuls\nuls-1.1.3\nuls\db-
module\db\src\main\java\io\nuls\db\constant\DBErrorCode.java
    ErrorCode DB_BATCH_CLOSE = ErrorCode.init("20015");
}

105:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\db\src\main\java\io\nuls\db\model\Entry.java

public class Entry<K, V> implements Comparable<K> {
    final K key;
    V value;
    Comparator<K> comparator;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public Entry(K key, V value, Comparator<K> comparator) {
        this.key = key;
        this.value = value;
        this.comparator = comparator;
    }

    public final K getKey()        { return key; }
    public final V getValue()      { return value; }
    @Override
    public final String toString() { return key + "=" + value; }

    @Override
    public final int hashCode() {
```

```
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    @Override
    public final boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (o instanceof Entry) {
            Entry<?,?> e = (Entry<?,?>)o;
            if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue())) {
                return true;
            }
        }
        return false;
    }

    @Override
    public int compareTo(K thatKey) {
        return comparator.compare(key, thatKey);
    }
}
```

106:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\db\src\main\java\io\nuls\db\model\ModelWrapper.java

```
    public ModelWrapper() {
    }

    public ModelWrapper(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}
```

107:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\db\src\main\java\io\nuls\db\module\AbstractDBModule.java

```
package io.nuls.db.module;


import io.nuls.db.constant.DBConstant;
import io.nuls.kernel.module.BaseModuleBootstrap;

public abstract class AbstractDBModule extends BaseModuleBootstrap {

    protected AbstractDBModule() {
        super(DBConstant.MODULE_ID_DB);
    }

}
```

108:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\db\src\main\java\io\nuls\db\service\BatchOperation.java

```
package io.nuls.db.service;

import io.nuls.kernel.model.Result;

public interface BatchOperation {

    /**
     *
     * Add or update operations.
     *
     * @param key
     * @param value
     * @return
     */
    Result put(byte[] key, byte[] value);

    /**
     *
     * Add or update the object
     *
     * @param key
     * @param value /Objects that need to be added or updated.
     * @return
```

```
     */
    <T> Result putModel(byte[] key, T value);

    /**
     *
     * Delete operation
     *
     * @param key
     * @return
     */
    Result delete(byte[] key);

    /**
     *
     * Perform batch operation
     *
     * @return
     */
    Result executeBatch();
}
```

109:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\db\src\main\java\io\nuls\db\service\DBService.java

```
package io.nuls.db.service;


import io.nuls.db.model.Entry;
import io.nuls.db.model.ModelWrapper;
import io.nuls.kernel.model.BaseNulsData;
import io.nuls.kernel.model.Result;

import java.util.Comparator;
import java.util.List;
import java.util.Set;

public interface DBService {

    /**
     *
     * Create a data area
     *
     * @param areaName
     * @return
```

```
     */
    Result createArea(String areaName);

    /**
     *
     * Deprecated method
     * @param areaName
     * @param cacheSize LevelDB/cacheSize hasn't been implemented.
     * @return
     */
    @Deprecated
    Result createArea(String areaName, Long cacheSize);

    /**
     * key
     * Create a data area for the custom key comparator.
     *
     * @param areaName
     * @param comparator key/Custom key comparator.
     * @return
     */
    Result createArea(String areaName, Comparator<byte[]> comparator);

    /**
     *
     * Deprecated method
     * @param areaName
     * @param cacheSize LevelDB/cacheSize hasn't been implemented by LevelDB in Java's
version.
     * @param comparator
     * @return
     */
    @Deprecated
    Result createArea(String areaName, Long cacheSize, Comparator<byte[]> comparator);


    /**
     * Area
     * Lists all Area names in the current database
     *
     * @return
     */
```

```
String[] listArea();

/**
 * key-value
 * Store key-value in bytes.
 *
 * @param area
 * @param key
 * @param value
 * @return
 */
Result put(String area, byte[] key, byte[] value);

/**
 *
 * Store the object
 *
 * @param area
 * @param key
 * @param value /Objects that need to be stored.
 * @param <T>
 * @return
 */
<T> Result putModel(String area, byte[] key, T value);

/**
 * keyvalue
 * Delete value according to key.
 *
 * @param area
 * @param key
 * @return
 */
Result delete(String area, byte[] key);

/**
 * keyvalue
 * Get value from the key.
 *
 * @param area
 * @param key
 * @return
```

```
 */
byte[] get(String area, byte[] key);

/**
 * keyclass
 * keyputModelvaluenull
 * Gets the specified object from the key and object class.
 * The premise is that this key is stored in a putModel, otherwise value is null.
 *
 * @param area
 * @param key
 * @param clazz class/Specifies the class of the object.
 * @param <T>
 * @return
 */
<T> T getModel(String area, byte[] key, Class<T> clazz);

/**
 * keyObject
 * Get the Object of Object from the key.
 *
 * param area
 * @param key
 * @return
 */
Object getModel(String area, byte[] key);

/**
 * key
 * Gets an unordered collection of all keys in the data area.
 *
 * @param area
 * @return
 */
Set<byte[]> keySet(String area);

/**
 * key
 * Gets an ordered collection of all keys in the data area.
 *
 * @param area
 * @return
```

```
     */
    List<byte[]> keyList(String area);

    /**
     * value
     * Gets an ordered collection of all values in the data area.
     *
     * @param area
     * @return
     */
    List<byte[]> valueList(String area);

    /**
     * key-value
     * Gets an unordered collection of all key-value in the data area.
     *
     * @param area
     * @return
     */
    Set<Entry<byte[], byte[]>> entrySet(String area);

    /**
     * key-value
     * Gets an ordered set of all key-values in the data area.
     *
     * @param area
     * @return
     */
    List<Entry<byte[], byte[]>> entryList(String area);


    /**
     * key-valuevalue
     * putModelvaluenull
     * Gets the ordered collection of all key-value in the data area and specifies the returned value object.
     * The premise is that the storage mode in this data area is the putModel, otherwise value is null.
     *
     * @param area
     * @param clazz class/Specifies the class of the object.
     * @param <T>
```

```
     * @return
     */
    <T> List<Entry<byte[], T>> entryList(String area, Class<T> clazz);

    /**
     * valuevalue
     * putModelvaluenull
     * Gets the ordered collection of all values in the data area and specifies the returned value object.
     * The premise is that the storage mode in this data area is the putModel, otherwise value is null.
     *
     * @param area
     * @param clazz class/Specifies the class of the object.
     * @param <T>
     * @return
     */
    <T> List<T> values(String area, Class<T> clazz);

    /**
     *
     * Specifies the batch add, delete, update operations in the data area.
     *
     * @param area
     * @return
     */
    BatchOperation createWriteBatch(String area);

    /**
     * Area
     *
     * @param areaName
     * @return
     */
    Result destroyArea(String areaName);

    Result clearArea(String area);

}
```

110:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-leveldb\src\main\java\io\nuls\db\manager\LevelDBManager.java

```
import io.nuls.core.tools.str.StringUtils;
import io.nuls.db.constant.DBErrorCode;
import io.nuls.db.model.Entry;
import io.nuls.db.model.ModelWrapper;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.model.Result;
import io.protostuff.LinkedBuffer;
import io.protostuff.ProtostuffIOUtil;
import io.protostuff.runtime.RuntimeSchema;
import org.iq80.leveldb.DB;
import org.iq80.leveldb.DBFactory;
import org.iq80.leveldb.DBIterator;
import org.iq80.leveldb.Options;
import org.iq80.leveldb.impl.Iq80DBFactory;

import java.io.File;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.URL;
import java.net.URLDecoder;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantLock;

import static io.nuls.core.tools.str.StringUtils.bytes;
import static io.nuls.db.constant.DBConstant.BASE_AREA_NAME;

public class LevelDBManager {

    private static int max;

    private static final ConcurrentHashMap<String, DB> AREAS = new ConcurrentHashMap<>();
    private static final ConcurrentHashMap<String, Comparator<byte[]>> AREAS_COMPARATOR
= new ConcurrentHashMap<>();

    private static final Map<Class, RuntimeSchema> SCHEMA_MAP = new
ConcurrentHashMap<>();

    private static final String BASE_DB_NAME = "leveldb";

    private static volatile boolean isInit = false;
```

```
    private static ReentrantLock lock = new ReentrantLock();

    private static String dataPath;

    public static int getMax() {
        return max;
    }

    public static String getBaseAreaName() {
        return BASE_AREA_NAME;
    }

    public static void init() throws Exception {
        synchronized (LevelDBManager.class) {
            if (!isInit) {
                isInit = true;
                File dir = loadDataPath();
                dataPath = dir.getPath();
                Log.info("LevelDBManager dataPath is " + dataPath);

                initSchema();
                initBaseDB(dataPath);

                File[] areaFiles = dir.listFiles();
                DB db;
                String dbPath = null;
                for (File areaFile : areaFiles) {
                    if (BASE_AREA_NAME.equals(areaFile.getName())) {
                        continue;
                    }
                    if (!areaFile.isDirectory()) {
                        continue;
                    }
                    try {
                        dbPath = areaFile.getPath() + File.separator + BASE_DB_NAME;
                        db = initOpenDB(dbPath);
                        if (db != null) {
                            AREAS.put(areaFile.getName(), db);
                        }
                    } catch (Exception e) {
                        Log.warn("load area failed, areaName: " + areaFile.getName() + ", dbPath: " +
dbPath, e);
```

```
            }

        }
    }
}

}

private static void initSchema() {
    RuntimeSchema schema = RuntimeSchema.createFrom(ModelWrapper.class);
    SCHEMA_MAP.put(ModelWrapper.class, schema);
}



/**
 * BASE_AREA
 * AreaAreaAreajava.lang.IllegalArgumentException: Expected user comparator leveldb.BytewiseComparator to match existing database comparator
 * AreacacheSizeAreacacheSize
 * Prioritize BASE_AREA.
 * Based data storage, for example, Area of custom comparator, the next time you start the database access and loaded it, otherwise, if the Area custom the comparator, the next time you restart the database, this Area will start failure, failure exception: java.lang.IllegalArgumentException: Expected user comparator leveldb. BytewiseComparator to match existing database comparator
 * the custom cacheSize of the Area will be retrieved and loaded next time the database is started, otherwise, the previous cacheSize setting will be lost when the existing Area is started.
 */
private static void initBaseDB(String dataPath) {
    if (AREAS.get(BASE_AREA_NAME) == null) {
        String baseAreaPath = dataPath + File.separator + BASE_AREA_NAME;
        File dir = new File(baseAreaPath);
        if (!dir.exists()) {
            dir.mkdir();
        }
        String filePath = baseAreaPath + File.separator + BASE_DB_NAME;
        try {
            DB db = openDB(filePath, true, null, null);
            AREAS.put(BASE_AREA_NAME, db);
        } catch (IOException e) {
            Log.error(e);
        }
```

```
    }
}

public static File loadDataPath() throws Exception {
    Properties properties = ConfigLoader.loadProperties("db_config.properties");
    String path = properties.getProperty("leveldb.datapath", "./data");
    String maxStr = properties.getProperty("leveldb.area.max", "50");
    try {
        max = Integer.parseInt(maxStr);
    } catch (Exception e) {
        //skip it
        max = 50;
    }
    File dir;
    String pathSeparator = System.getProperty("path.separator");
    String unixPathSeparator = ":";
    String rootPath;
    if (unixPathSeparator.equals(pathSeparator)) {
        rootPath = "/";
        if (path.startsWith(rootPath)) {
            dir = new File(path);
        } else {
            dir = new File(genAbsolutePath(path));
        }
    } else {
        rootPath = "^[c-zC-Z]:.*";
        if (path.matches(rootPath)) {
            dir = new File(path);
        } else {
            dir = new File(genAbsolutePath(path));
        }
    }

    if (!dir.exists()) {
        dir.mkdirs();
    }
    return dir;
}

private static String genAbsolutePath(String path) {
    String[] paths = path.split("/|\\\\");
    URL resource = ClassLoader.getSystemClassLoader().getResource("");
```

```
    String classPath = resource.getPath();
    File file = null;
    try {
        file = new File(URLDecoder.decode(classPath, "UTF-8"));
    } catch (UnsupportedEncodingException e) {
        Log.error(e);
        file = new File(classPath);
    }
    String resultPath = null;
    boolean isFileName = false;
    for (String p : paths) {
        if (StringUtils.isBlank(p)) {
            continue;
        }
        if (!isFileName) {
            if ("..".equals(p)) {
                file = file.getParentFile();
            } else if (".".equals(p)) {
                continue;
            } else {
                isFileName = true;
                resultPath = file.getPath() + File.separator + p;
            }
        } else {
            resultPath += File.separator + p;
        }
    }
    return resultPath;
}

public static Result createArea(String areaName) {
    return createArea(areaName, null, null);
}

public static Result createArea(String areaName, Long cacheSize) {
    return createArea(areaName, cacheSize, null);
}

public static Result createArea(String areaName, Comparator<byte[]> comparator) {
    return createArea(areaName, null, comparator);
}
```

```
    public static Result createArea(String areaName, Long cacheSize, Comparator<byte[]>
comparator) {
        lock.lock();
        try {
            if (StringUtils.isBlank(areaName)) {
                return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
            }
            if (AREAS.containsKey(areaName)) {
                return Result.getFailed(DBErrorCode.DB_AREA_EXIST);
            }
            // prevent too many areas
            if (AREAS.size() > (max - 1)) {
                return Result.getFailed(DBErrorCode.DB_AREA_CREATE_EXCEED_LIMIT);
            }
            if (StringUtils.isBlank(dataPath) || !checkPathLegal(areaName)) {
                return Result.getFailed(DBErrorCode.DB_AREA_CREATE_PATH_ERROR);
            }
            Result result;
            try {
                File dir = new File(dataPath + File.separator + areaName);
                if (!dir.exists()) {
                    dir.mkdir();
                }
                String filePath = dataPath + File.separator + areaName + File.separator +
BASE_DB_NAME;
                DB db = openDB(filePath, true, cacheSize, comparator);
                AREAS.put(areaName, db);
                result = Result.getSuccess();
            } catch (Exception e) {
                Log.error("error create area: " + areaName, e);
                result = Result.getFailed(DBErrorCode.DB_AREA_CREATE_ERROR);
            }
            return result;
        } finally {
            lock.unlock();
        }
    }

    public static DB getArea(String areaName) {
        return AREAS.get(areaName);
    }
```

```
    public static Result destroyArea(String areaName) {
        if (!baseCheckArea(areaName)) {
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if (StringUtils.isBlank(dataPath) || !checkPathLegal(areaName)) {
            return Result.getFailed(DBErrorCode.DB_AREA_CREATE_PATH_ERROR);
        }
        Result result;
        try {
            DB db = AREAS.remove(areaName);
            db.close();
            File dir = new File(dataPath + File.separator + areaName);
            if (!dir.exists()) {
                return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
            }
            String filePath = dataPath + File.separator + areaName + File.separator +
BASE_DB_NAME;
            destroyDB(filePath);
            AREAS_COMPARATOR.remove(areaName);
            delete(BASE_AREA_NAME, bytes(areaName + "-comparator"));
            delete(BASE_AREA_NAME, bytes(areaName + "-cacheSize"));
            result = Result.getSuccess();
        } catch (Exception e) {
            Log.error("error destroy area: " + areaName, e);
            result = Result.getFailed(DBErrorCode.DB_AREA_DESTROY_ERROR);
        }
        return result;
    }

    private static void destroyDB(String dbPath) throws IOException {
        File file = new File(dbPath);
        Options options = new Options();
        DBFactory factory = Iq80DBFactory.factory;
        factory.destroy(file, options);
    }

    /**
     * close all area
     *
     */
    public static void close() {
        Set<Map.Entry<String, DB>> entries = AREAS.entrySet();
```

```
    for (Map.Entry<String, DB> entry : entries) {
        try {
            AREAS.remove(entry.getKey());
            AREAS_COMPARATOR.remove(entry.getKey());
            entry.getValue().close();
        } catch (Exception e) {
            Log.warn("close leveldb error", e);
        }
    }
}

/**
 * close a area
 *
 */
public static void closeArea(String area) {
    try {
        AREAS_COMPARATOR.remove(area);
        DB db = AREAS.remove(area);
        db.close();
    } catch (IOException e) {
        Log.warn("close leveldb area error:" + area, e);
    }
}



/**
 * @param dbPath
 * @return
 * @throws IOException
 */
private static DB initOpenDB(String dbPath) throws IOException {
    File checkFile = new File(dbPath + File.separator + "CURRENT");
    if (!checkFile.exists()) {
        return null;
    }
    Options options = new Options().createIfMissing(false);

    /*
     * Area
     * AreacacheSizeAreacacheSize
     * Area of custom comparator, you start the database access and loaded it
```

```
         * the custom cacheSize of the Area will be retrieved and loaded on the database is started, otherwise, the previous cacheSize setting will be lost when the existing Area is started.
         */
        String areaName = getAreaNameFromDbPath(dbPath);
        Comparator comparator = getModel(BASE_AREA_NAME, bytes(areaName + "-comparator"), Comparator.class);
        if (comparator != null) {
            AREAS_COMPARATOR.put(areaName, comparator);
        }
        Long cacheSize = getModel(BASE_AREA_NAME, bytes(areaName + "-cacheSize"), Long.class);
        if (cacheSize != null) {
            options.cacheSize(cacheSize);
        }
        File file = new File(dbPath);
        DBFactory factory = Iq80DBFactory.factory;
        return factory.open(file, options);
    }

    /**
     *
     * Areaarea
     * AreacacheSizeareacacheSizeAreacacheSize
     * load database
     * If the area custom comparator, save area define the comparator, the next time you start the database access and loaded it
     * If the area custom cacheSize, save the area's custom cacheSize, get and load it the next time you start the database, or you'll lose the cacheSize setting before starting the existing area.
     */
    private static DB openDB(String dbPath, boolean createIfMissing, Long cacheSize, Comparator<byte[]> comparator) throws IOException {
        File file = new File(dbPath);
        String areaName = getAreaNameFromDbPath(dbPath);
        Options options = new Options().createIfMissing(createIfMissing);
        if (cacheSize != null) {
            putModel(BASE_AREA_NAME, bytes(areaName + "-cacheSize"), cacheSize);
            options.cacheSize(cacheSize);
        }
        if (comparator != null) {
            putModel(BASE_AREA_NAME, bytes(areaName + "-comparator"), comparator);
            AREAS_COMPARATOR.put(areaName, comparator);
        }
```

```
    DBFactory factory = Iq80DBFactory.factory;
    return factory.open(file, options);
}

private static String getAreaNameFromDbPath(String dbPath) {
    int end = dbPath.lastIndexOf(File.separator);
    int start = dbPath.lastIndexOf(File.separator, end - 1) + 1;
    return dbPath.substring(start, end);
}

private static boolean checkPathLegal(String areaName) {
    if (StringUtils.isBlank(areaName)) {
        return false;
    }
    String regex = "^[a-zA-Z0-9_\\-]+$";
    return areaName.matches(regex);
}

private static boolean baseCheckArea(String areaName) {
    if (StringUtils.isBlank(areaName) || !AREAS.containsKey(areaName)) {
        return false;
    }
    return true;
}

public static String[] listArea() {
    int i = 0;
    Enumeration<String> keys = AREAS.keys();
    String[] areas = new String[AREAS.size()];
    int length = areas.length;
    while (keys.hasMoreElements()) {
        areas[i++] = keys.nextElement();
        // thread safe, prevent java.lang.ArrayIndexOutOfBoundsException
        if (i == length) {
            break;
        }
    }
    return areas;
}

public static Result put(String area, byte[] key, byte[] value) {
    if (!baseCheckArea(area)) {
```

```
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if (key == null || value == null) {
            return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
        }
        try {
            DB db = AREAS.get(area);
            db.put(key, value);
            return Result.getSuccess();
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        }
    }

    /**
     * /Deprecated method
     */
    @Deprecated
    public static Result put(String area, String key, String value) {
        if (!baseCheckArea(area)) {
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if (StringUtils.isBlank(key) || StringUtils.isBlank(value)) {
            return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
        }
        try {
            DB db = AREAS.get(area);
            db.put(bytes(key), bytes(value));
            return Result.getSuccess();
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        }
    }

    /**
     * /Deprecated method
     */
    @Deprecated
    public static Result put(String area, byte[] key, String value) {
        if (!baseCheckArea(area)) {
```

```
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if (key == null || StringUtils.isBlank(value)) {
            return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
        }
        try {
            DB db = AREAS.get(area);
            db.put(key, bytes(value));
            return Result.getSuccess();
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        }
    }

    /**
     * /Deprecated method
     */
    @Deprecated
    public static <T> Result putModel(String area, String key, T value) {
        return putModel(area, bytes(key), value);
    }

    public static <T> Result putModel(String area, byte[] key, T value) {
        if (!baseCheckArea(area)) {
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if (key == null || value == null) {
            return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
        }
        try {
            byte[] bytes = getModelSerialize(value);
            return put(area, key, bytes);
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        }
    }

    public static <T> byte[] getModelSerialize(T value) {
        if (SCHEMA_MAP.get(ModelWrapper.class) == null) {
            RuntimeSchema schema = RuntimeSchema.createFrom(ModelWrapper.class);
```

```
        SCHEMA_MAP.put(ModelWrapper.class, schema);
    }
    RuntimeSchema schema = SCHEMA_MAP.get(ModelWrapper.class);
    ModelWrapper modelWrapper = new ModelWrapper(value);
    LinkedBuffer buffer = LinkedBuffer.allocate(LinkedBuffer.DEFAULT_BUFFER_SIZE);
    final byte[] bytes;
    try {
        bytes = ProtostuffIOUtil.toByteArray(modelWrapper, schema, buffer);
    } finally {
        buffer.clear();
    }
    return bytes;
}

/**
 * /Deprecated method
 */
@Deprecated
public static Result delete(String area, String key) {
    if (!baseCheckArea(area)) {
        return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
    }
    if (StringUtils.isBlank(key)) {
        return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
    }
    try {
        DB db = AREAS.get(area);
        db.delete(bytes(key));
        return Result.getSuccess();
    } catch (Exception e) {
        Log.error(e);
        return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
    }
}

public static Result delete(String area, byte[] key) {
    if (!baseCheckArea(area)) {
        return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
    }
    if (key == null) {
        return Result.getFailed(KernelErrorCode.NULL_PARAMETER);
    }
```

```
    try {
        DB db = AREAS.get(area);
        db.delete(key);
        return Result.getSuccess();
    } catch (Exception e) {
        Log.error(e);
        return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
    }
}

/**
 * /Deprecated method
 */
@Deprecated
public static byte[] get(String area, String key) {
    if (!baseCheckArea(area)) {
        return null;
    }
    if (StringUtils.isBlank(key)) {
        return null;
    }
    try {
        DB db = AREAS.get(area);
        return db.get(bytes(key));
    } catch (Exception e) {
        return null;
    }
}

public static byte[] get(String area, byte[] key) {
    if (!baseCheckArea(area)) {
        return null;
    }
    if (key == null) {
        return null;
    }
    try {
        DB db = AREAS.get(area);
        return db.get(key);
    } catch (Exception e) {
        return null;
    }
```

```
}

/**
 * /Deprecated method
 */
@Deprecated
public static Object getModel(String area, String key) {
    return getModel(area, bytes(key));
}

public static Object getModel(String area, byte[] key) {
    return getModel(area, key, null);
}

/**
 * /Deprecated method
 */
@Deprecated
public static <T> T getModel(String area, String key, Class<T> clazz) {
    return getModel(area, bytes(key), clazz);
}

public static <T> T getModel(String area, byte[] key, Class<T> clazz) {
    if (!baseCheckArea(area)) {
        return null;
    }
    if (key == null) {
        return null;
    }
    try {
        DB db = AREAS.get(area);
        byte[] bytes = db.get(key);
        if (bytes == null) {
            return null;
        }
        RuntimeSchema schema = SCHEMA_MAP.get(ModelWrapper.class);
        ModelWrapper model = new ModelWrapper();
        ProtostuffIOUtil.mergeFrom(bytes, model, schema);
        if (clazz != null && model.getT() != null) {
            return clazz.cast(model.getT());
        }
        return (T) model.getT();
```

```
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

public static Set<byte[]> keySet(String area) {
    if (!baseCheckArea(area)) {
        return null;
    }
    DBIterator iterator = null;
    Set<byte[]> keySet;
    try {
        DB db = AREAS.get(area);
        keySet = new HashSet<>();
        iterator = db.iterator();
        for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
            keySet.add(iterator.peekNext().getKey());
        }
        return keySet;
    } catch (Exception e) {
        Log.error(e);
        return null;
    } finally {
        // Make sure you close the iterator to avoid resource leaks.
        if (iterator != null) {
            try {
                iterator.close();
            } catch (IOException e) {
                //skip it
            }
        }
    }
}

public static List<byte[]> keyList(String area) {
    if (!baseCheckArea(area)) {
        return null;
    }
    DBIterator iterator = null;
    List<byte[]> keyList;
    try {
```

```
            DB db = AREAS.get(area);
            keyList = new ArrayList<>();
            iterator = db.iterator();
            String key;
            for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
                keyList.add(iterator.peekNext().getKey());
            }
            Comparator<byte[]> comparator = AREAS_COMPARATOR.get(area);
            if (comparator != null) {
                keyList.sort(comparator);
            }
            return keyList;
        } catch (Exception e) {
            Log.error(e);
            return null;
        } finally {
            // Make sure you close the iterator to avoid resource leaks.
            if (iterator != null) {
                try {
                    iterator.close();
                } catch (IOException e) {
                    //skip it
                }
            }
        }
    }

    public static Set<Entry<byte[], byte[]>> entrySet(String area) {
        if (!baseCheckArea(area)) {
            return null;
        }
        DBIterator iterator = null;
        Set<Entry<byte[], byte[]>> entrySet;
        try {
            DB db = AREAS.get(area);
            entrySet = new HashSet<>();
            iterator = db.iterator();
            byte[] key, bytes;
            Map.Entry<byte[], byte[]> entry;
            for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
                entry = iterator.peekNext();
                key = entry.getKey();
```

```
            bytes = entry.getValue();
            entrySet.add(new Entry<byte[], byte[]>(key, bytes));
        }
    } catch (Exception e) {
        Log.error(e);
        return null;
    } finally {
        // Make sure you close the iterator to avoid resource leaks.
        if (iterator != null) {
            try {
                iterator.close();
            } catch (IOException e) {
                //skip it
            }
        }
    }
    return entrySet;
}

public static List<Entry<byte[], byte[]>> entryList(String area) {
    if (!baseCheckArea(area)) {
        return null;
    }
    DBIterator iterator = null;
    List<Entry<byte[], byte[]>> entryList;
    try {
        DB db = AREAS.get(area);
        entryList = new ArrayList<>();
        iterator = db.iterator();
        byte[] key, bytes;
        Map.Entry<byte[], byte[]> entry;
        Comparator<byte[]> comparator = AREAS_COMPARATOR.get(area);
        for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
            entry = iterator.peekNext();
            key = entry.getKey();
            bytes = entry.getValue();
            entryList.add(new Entry<byte[], byte[]>(key, bytes, comparator));
        }
        //
        if (comparator != null) {
            entryList.sort(new Comparator<Entry<byte[], byte[]>>() {
                @Override
```

```
                public int compare(Entry<byte[], byte[]> o1, Entry<byte[], byte[]> o2) {
                    return o1.compareTo(o2.getKey());
                }
            });
        }
    } catch (Exception e) {
        Log.error(e);
        return null;
    } finally {
        // Make sure you close the iterator to avoid resource leaks.
        if (iterator != null) {
            try {
                iterator.close();
            } catch (IOException e) {
                //skip it
            }
        }
    }
    return entryList;
}

public static <T> List<Entry<byte[], T>> entryList(String area, Class<T> clazz) {
    if (!baseCheckArea(area)) {
        return null;
    }
    DBIterator iterator = null;
    List<Entry<byte[], T>> entryList;
    try {
        DB db = AREAS.get(area);
        entryList = new ArrayList<>();
        iterator = db.iterator();
        byte[] key;
        Map.Entry<byte[], byte[]> entry;
        Comparator<byte[]> comparator = AREAS_COMPARATOR.get(area);
        T t;
        for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
            t = null;
            entry = iterator.peekNext();
            key = entry.getKey();
            t = getModel(area, entry.getKey(), clazz);
            entryList.add(new Entry<byte[], T>(key, t, comparator));
        }
```

```
        //
        if (comparator != null) {
            entryList.sort(new Comparator<Entry<byte[], T>>() {
                @Override
                public int compare(Entry<byte[], T> o1, Entry<byte[], T> o2) {
                    return o1.compareTo(o2.getKey());
                }
            });
        }
    } catch (Exception e) {
        Log.error(e);
        return null;
    } finally {
        // Make sure you close the iterator to avoid resource leaks.
        if (iterator != null) {
            try {
                iterator.close();
            } catch (IOException e) {
                //skip it
            }
        }
    }
    return entryList;
}

public static <T> List<T> values(String area, Class<T> clazz) {
    if (!baseCheckArea(area)) {
        return null;
    }
    try {
        Comparator<byte[]> comparator = AREAS_COMPARATOR.get(area);
        if (comparator == null) {
            return valuesInner(area, clazz);
        } else {
            List<Entry<byte[], T>> entryList = entryList(area, clazz);
            List<T> resultList = new ArrayList<>();
            if (entryList != null) {
                entryList.stream().forEach(entry -> resultList.add(entry.getValue()));
            }
            return resultList;
        }
    } catch (Exception e) {
```

```
            Log.error(e);
            return null;
        }
    }

    private static <T> List<T> valuesInner(String area, Class<T> clazz) {
        if (!baseCheckArea(area)) {
            return null;
        }
        DBIterator iterator = null;
        List<T> list;
        try {
            DB db = AREAS.get(area);
            list = new ArrayList<>();
            iterator = db.iterator();
            Map.Entry<byte[], byte[]> entry;
            T t;
            for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
                t = null;
                entry = iterator.peekNext();
                t = getModel(area, entry.getKey(), clazz);
                list.add(t);
            }
        } catch (Exception e) {
            Log.error(e);
            return null;
        } finally {
            // Make sure you close the iterator to avoid resource leaks.
            if (iterator != null) {
                try {
                    iterator.close();
                } catch (IOException e) {
                    //skip it
                }
            }
        }
        return list;
    }

    public static List<byte[]> valueListInner(String area) {
        if (!baseCheckArea(area)) {
            return null;
```

```
    }
    DBIterator iterator = null;
    List<byte[]> list;
    try {
        DB db = AREAS.get(area);
        list = new ArrayList<>();
        iterator = db.iterator();
        for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
            list.add(iterator.peekNext().getValue());
        }
    } catch (Exception e) {
        Log.error(e);
        return null;
    } finally {
        // Make sure you close the iterator to avoid resource leaks.
        if (iterator != null) {
            try {
                iterator.close();
            } catch (IOException e) {
                //skip it
            }
        }
    }
    return list;
}

public static List<byte[]> valueList(String area) {
    if (!baseCheckArea(area)) {
        return null;
    }
    try {
        Comparator<byte[]> comparator = AREAS_COMPARATOR.get(area);
        if (comparator == null) {
            return valueListInner(area);
        } else {
            List<Entry<byte[], byte[]>> entryList = entryList(area);
            List<byte[]> resultList = new ArrayList<>();
            if (entryList != null) {
                entryList.stream().forEach(entry -> resultList.add(entry.getValue()));
            }
            return resultList;
        }
```

```
        } catch (Exception e) {
            Log.error(e);
            return null;
        }
    }

    public static Result clearArea(String area) {
        if (!baseCheckArea(area)) {
            return Result.getFailed();
        }
        try {
            return destroyArea(area);
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed();
        }

        /*DBIterator iterator = null;
        try {
            DB db = AREAS.get(area);
            iterator = db.iterator();
            for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
               db.delete(iterator.peekNext().getKey());
            }
            return Result.getSuccess();
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed();
        } finally {
            // Make sure you close the iterator to avoid resource leaks.
            if (iterator != null) {
                try {
                    iterator.close();
                } catch (IOException e) {
                    //skip it
                }
            }
        }*/
    }
}

111:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-
```

leveldb\src\main\java\io\nuls\db\module\impl\LevelDbModuleBootstrap.java

```
import io.nuls.db.module.AbstractDBModule;
import org.iq80.leveldb.DBException;

public class LevelDbModuleBootstrap extends AbstractDBModule {

    @Override
    public void init() {
        try {
            initLevelDBStorage();
        } catch (Exception e) {
            Log.error(e);
            throw new DBException(e);
        }
    }

    private void initLevelDBStorage() throws Exception {
//        LevelDBManager.init();
    }

    @Override
    public void start() {

    }

    @Override
    public void shutdown() {
        LevelDBManager.close();
    }

    @Override
    public void destroy() {
        LevelDBManager.close();
    }

    @Override
    public String getInfo() {
        StringBuilder str = new StringBuilder();
        str.append("moduleName:");
        str.append(getModuleName());
        str.append(",moduleStatus:");
        str.append(getStatus());
```

```
        str.append(",ThreadCount:");
        return str.toString();
    }
}
```

112:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-leveldb\src\main\java\io\nuls\db\service\impl\BatchOperationImpl.java

```
import io.nuls.db.manager.LevelDBManager;
import io.nuls.db.service.BatchOperation;
import io.nuls.kernel.model.Result;
import org.iq80.leveldb.DB;
import org.iq80.leveldb.WriteBatch;

import java.io.IOException;

public class BatchOperationImpl implements BatchOperation {

    private static final Result FAILED_NULL =
Result.getFailed(DBErrorCode.NULL_PARAMETER);
    private static final Result SUCCESS = Result.getSuccess();
    private static final Result FAILED_BATCH_CLOSE =
Result.getFailed(DBErrorCode.DB_BATCH_CLOSE);
    private String area;
    private DB db;
    private WriteBatch batch;
    private volatile boolean isClose = false;

    BatchOperationImpl(String area) {
        this.area = area;
        db = LevelDBManager.getArea(area);
        if(db != null) {
            batch = db.createWriteBatch();
        }
    }

    public Result checkBatch() {
        if(db == null) {
            return Result.getFailed(DBErrorCode.DB_AREA_NOT_EXIST);
        }
        if(batch == null) {
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        }
```

```
    return SUCCESS;
}

@Override
public Result put(byte[] key, byte[] value) {
    if(key == null || value == null) {
        return FAILED_NULL;
    }
    batch.put(key, value);
    return SUCCESS;
}

@Override
public <T> Result putModel(byte[] key, T value) {
    if(key == null || value == null) {
        return FAILED_NULL;
    }
    byte[] bytes = LevelDBManager.getModelSerialize(value);
    return put(key, bytes);
}

@Override
public Result delete(byte[] key) {
    if(key == null) {
        return FAILED_NULL;
    }
    batch.delete(key);
    return SUCCESS;
}

private void close() {
    this.isClose = true;
}

private boolean checkClose() {
    return isClose;
}

@Override
public Result executeBatch() {
    //
    if(checkClose()) {
```

```
            return FAILED_BATCH_CLOSE;
        }
        try {
            db.write(batch);
        } catch (Exception e) {
            Log.error(e);
            return Result.getFailed(DBErrorCode.DB_UNKOWN_EXCEPTION);
        } finally {
            // Make sure you close the batch to avoid resource leaks.
            // LevelDBclose,
            if(batch != null) {
                try {
                    this.close();
                    batch.close();
                } catch (IOException e) {
                    // skip it
                }
            }
        }
        return SUCCESS;
    }
}
```

113:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-leveldb\src\main\java\io\nuls\db\service\impl\LevelDBServiceImpl.java

```
import io.nuls.db.manager.LevelDBManager;
import io.nuls.db.model.Entry;
import io.nuls.db.service.BatchOperation;
import io.nuls.db.service.DBService;
import io.nuls.kernel.lite.annotation.Service;
import io.nuls.kernel.model.Result;

import java.util.Comparator;
import java.util.List;
import java.util.Set;

@Service
public class LevelDBServiceImpl implements DBService {

    public LevelDBServiceImpl() {
        try {
            LevelDBManager.init();
```

```
    } catch (Exception e) {
        //skip it
    }
}

@Override
public Result createArea(String areaName) {
    return LevelDBManager.createArea(areaName);
}

@Override
public Result createArea(String areaName, Long cacheSize) {
    return LevelDBManager.createArea(areaName, cacheSize);
}

@Override
public Result createArea(String areaName, Comparator<byte[]> comparator) {
    return LevelDBManager.createArea(areaName, comparator);
}

@Override
public Result createArea(String areaName, Long cacheSize, Comparator<byte[]> comparator) {
    return LevelDBManager.createArea(areaName, cacheSize, comparator);
}

@Override
public String[] listArea() {
    return LevelDBManager.listArea();
}

@Override
public Result put(String area, byte[] key, byte[] value) {
    return LevelDBManager.put(area, key, value);
}

@Override
public <T> Result putModel(String area, byte[] key, T value) {
    return LevelDBManager.putModel(area, key, value);
}

@Override
public Result delete(String area, byte[] key) {
```

```
    return LevelDBManager.delete(area, key);
}

@Override
public byte[] get(String area, byte[] key) {
    return LevelDBManager.get(area, key);
}

@Override
public <T> T getModel(String area, byte[] key, Class<T> clazz) {
    return LevelDBManager.getModel(area, key, clazz);
}

@Override
public Object getModel(String area, byte[] key) {
    return LevelDBManager.getModel(area, key);
}

@Override
public Set<byte[]> keySet(String area) {
    return LevelDBManager.keySet(area);
}

@Override
public List<byte[]> keyList(String area) {
    return LevelDBManager.keyList(area);
}

@Override
public List<byte[]> valueList(String area) {
    return LevelDBManager.valueList(area);
}

@Override
public Set<Entry<byte[], byte[]>> entrySet(String area) {
    return LevelDBManager.entrySet(area);
}

@Override
public List<Entry<byte[], byte[]>> entryList(String area) {
    return LevelDBManager.entryList(area);
}
```

```
    @Override
    public <T> List<Entry<byte[], T>> entryList(String area, Class<T> clazz) {
        return LevelDBManager.entryList(area, clazz);
    }

    @Override
    public <T> List<T> values(String area, Class<T> clazz) {
        return LevelDBManager.values(area, clazz);
    }

    @Override
    public BatchOperation createWriteBatch(String area) {
        if(StringUtils.isBlank(area)) {
            return null;
        }
        BatchOperationImpl batchOperation = new BatchOperationImpl(area);
        Result result = batchOperation.checkBatch();
        if(result.isFailed()) {
            Log.error("DB batch create error: " + result.getMsg());
            return null;
        }
        return batchOperation;
    }

    @Override
    public Result destroyArea(String area) {
        return LevelDBManager.destroyArea(area);
    }

    /**
     * Area
     */
    @Override
    public Result clearArea(String area) {
        return LevelDBManager.clearArea(area);
    }
}
```

114:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-leveldb\src\test\java\io\nuls\db\entity\DBTestEntity.java

```
 *
```

```
 */

package io.nuls.db.entity;

import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;

/**
 * Created by ln on 2018/5/6.
 */
public class DBTestEntity extends Transaction {

    public DBTestEntity() {
        super(0);
    }

    @Override
    protected TransactionLogicData parseTxData(NulsByteBuffer byteBuffer) {
        // todo auto-generated method stub
        return null;
    }

    @Override
    public String getInfo(byte[] address) {
        // todo auto-generated method stub
        return null;
    }
}

115:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-
leveldb\src\test\java\io\nuls\db\entity\TestTransaction.java
 */

package io.nuls.db.entity;

import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;

/**
```

```
 * @author: Niels Wang
 * @date: 2018/7/7
 */
public class TestTransaction extends Transaction {

    public TestTransaction( ) {
        super(100);
    }

    @Override
    protected TransactionLogicData parseTxData(NulsByteBuffer byteBuffer) throws NulsException
{
        return null;
    }

    @Override
    public String getInfo(byte[] address) {
        return null;
    }
}

116:F:\git\coin\nuls\nuls-1.1.3\nuls\db-module\leveldb\db-
leveldb\src\test\java\io\nuls\db\service\LevelDBServiceTest.java
 *
 */

package io.nuls.db.service;

import io.nuls.core.tools.crypto.ECKey;
import io.nuls.core.tools.log.Log;
import io.nuls.db.constant.DBErrorCode;
import io.nuls.db.entity.DBTestEntity;
import io.nuls.db.manager.LevelDBManager;
import io.nuls.db.model.Entry;
import io.nuls.db.service.impl.LevelDBServiceImpl;
import io.nuls.kernel.cfg.NulsConfig;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.*;

import io.nuls.kernel.script.SignatureUtil;
import io.nuls.kernel.utils.AddressTool;
```

```
import org.iq80.leveldb.impl.Iq80DBFactory;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;
import java.util.*;

import static io.nuls.db.manager.LevelDBManager.*;
import static org.iq80.leveldb.impl.Iq80DBFactory.asString;
import static org.iq80.leveldb.impl.Iq80DBFactory.bytes;
import static org.junit.Assert.*;

/**
 * Created by ln on 2018/5/6.
 */
public class LevelDBServiceTest {

    private static DBService dbService;

    private static String areaName = "transaction";

    private static String area;
    private static String key;

    @BeforeClass
    public static void init() {
        dbService = new LevelDBServiceImpl();
        dbService.createArea(areaName);
        area = "pierre-test";
        key = "testkey";
        createArea(area);
    }

    private static void setCommonFields(Transaction tx) {
        tx.setTime(System.currentTimeMillis());
        tx.setBlockHeight(1);
        tx.setRemark("for test".getBytes());
    }
```

```
    private static void signTransaction(Transaction tx, ECKey ecKey) throws IOException {
        NulsDigestData hash = null;
        hash = NulsDigestData.calcDigestData(tx.serializeForHash());
        tx.setHash(hash);
        SignatureUtil.createSignatureByEckey(tx, ecKey);
    }

    private static DBTestEntity createTransferTransaction(byte[] coinKey, Na na, long index) throws
IOException {
        ECKey ecKey1 = new ECKey();
        ECKey ecKey2 = new ECKey();
        DBTestEntity tx = new DBTestEntity();
        setCommonFields(tx);
        tx.setTime(index);
        CoinData coinData = new CoinData();
        List<Coin> fromList = new ArrayList<>();
        fromList.add(new Coin(coinKey, Na.parseNuls(10001), 0));
        coinData.setFrom(fromList);
        List<Coin> toList = new ArrayList<>();
        toList.add(new Coin(AddressTool.getAddress(ecKey2.getPubKey()), Na.parseNuls(10000),
1000));
        coinData.setTo(toList);
        tx.setCoinData(coinData);
        signTransaction(tx, ecKey1);
        return tx;
    }

    @Test
    public void testPerformanceTesting() throws IOException {

        long time = System.currentTimeMillis();

        long maxCount = 10000;
        for (long i = 0; i < maxCount; i++) {
            DBTestEntity entity = createTransferTransaction(null, Na.ZERO, i);

            byte[] key = ("entitySerialize" + i).getBytes(StandardCharsets.UTF_8);
            byte[] value = entity.serialize();

            Result result = dbService.put(areaName, key, value);
            assertNotNull(result);
```

```
            assertTrue(result.isSuccess());
        }

        System.out.println("Save " + maxCount + " transaction time-consuming : " +
(System.currentTimeMillis() - time) + " ms");

        time = System.currentTimeMillis();
        long getCount = 1000;

        System.out.println("Test random performance of " + getCount + " data …");
        for (long i = 0; i < getCount; i++) {

            long index = (long) (Math.random() * maxCount);

            byte[] resultBytes = dbService.get(areaName, ("entitySerialize" +
index).getBytes(StandardCharsets.UTF_8));
            assertNotNull(resultBytes);

            DBTestEntity e = new DBTestEntity();
            try {
                e.parse(resultBytes,0);
            } catch (NulsException e1) {
                Log.error(e1);
            }
            assertEquals(e.getTime(), index);
        }
//        C:\workspace\nuls_v2\db-module\leveldb\db-leveldb\target\test-classes\data\test\pierre-test-
15
//        C:\workspace\nuls_v2\db-module\leveldb\db-leveldb\target\test-classes\data\test\pierre-test-
15\leveldb
        System.out.println("It takes " + (System.currentTimeMillis() - time) + " ms to randomly acquire
" + getCount + " data");
        LevelDBManager.destroyArea(areaName);
    }

    @Test
    public void testBatch() {
        String area = "testBatch";
        dbService.createArea(area);
        BatchOperation batch = dbService.createWriteBatch(area);
        batch.put(bytes("Tampa"), bytes("green"));
        batch.put(bytes("London"), bytes("red"));
```

```
        batch.put(bytes("London1"), bytes("red1"));
        batch.put(bytes("London2"), bytes("red2"));
        batch.put(bytes("Qweqwe"), bytes("blue"));
        batch.delete(bytes("Qweqwe"));
        batch.delete(bytes("Qwe123qwe"));
        batch.executeBatch();

        List<Entry<byte[], byte[]>> entries = dbService.entryList(area);
        entries.stream().forEach(entry -> {
            System.out.print("[" + entry.getKey() + "=" + asString(entry.getValue()) + "], ");
        });
        System.out.println();

        Assert.assertEquals("green", asString(dbService.get(area, bytes("Tampa"))));
        Assert.assertEquals("red", asString(dbService.get(area, bytes("London"))));
        Assert.assertEquals("red1", asString(dbService.get(area, bytes("London1"))));
        Assert.assertEquals("red2", asString(dbService.get(area, bytes("London2"))));
        Assert.assertNull(dbService.get(area, bytes("Qweqwe")));

        //
        Result result = batch.executeBatch();
        Assert.assertTrue(result.isFailed());
        Assert.assertEquals(DBErrorCode.DB_BATCH_CLOSE.getCode(),
result.getErrorCode().getCode());
        LevelDBManager.destroyArea(area);
    }

    @Test
    public void testBatchModel() {
        String area = "testBatchModel";
        dbService.createArea(area);
        BatchOperation batch = dbService.createWriteBatch(area);
        DBTestEntity entity = new DBTestEntity();
        entity.setType(11111);
        batch.putModel(bytes("entity1"), entity);
        entity = new DBTestEntity();
        entity.setType(22222);
        batch.putModel(bytes("entity2"), entity);
        entity = new DBTestEntity();
        entity.setType(33333);
        batch.putModel(bytes("entity3"), entity);
        entity = new DBTestEntity();
```

```
        entity.setType(44444);
        batch.putModel(bytes("entity4"), entity);
        entity = new DBTestEntity();
        entity.setType(55555);
        batch.putModel(bytes("entity5"), entity);
        batch.executeBatch();
        List<DBTestEntity> list = dbService.values(area, DBTestEntity.class);
        list.stream().forEach(dbTestEntity -> {
            System.out.println("[" + dbTestEntity.toString() + "=" + dbTestEntity.getType() + "], ");
        });
        System.out.println();
        Assert.assertEquals(11111, dbService.getModel(area, bytes("entity1"),
DBTestEntity.class).getType());
        Assert.assertEquals(22222, dbService.getModel(area, bytes("entity2"),
DBTestEntity.class).getType());
        Assert.assertEquals(33333, dbService.getModel(area, bytes("entity3"),
DBTestEntity.class).getType());
        Assert.assertEquals(44444, dbService.getModel(area, bytes("entity4"),
DBTestEntity.class).getType());
        Assert.assertEquals(55555, dbService.getModel(area, bytes("entity5"),
DBTestEntity.class).getType());


        batch = dbService.createWriteBatch(area);
        batch.delete(bytes("entity4"));
        batch.delete(bytes("entity5"));
        batch.executeBatch();
        List<Entry<byte[], DBTestEntity>> entries = dbService.entryList(area, DBTestEntity.class);
        entries.stream().forEach(entry -> {
            System.out.println("[" + asString(entry.getKey()) + "=" + entry.getValue().getType() + "], ");
        });
        System.out.println();
        Assert.assertEquals(11111, dbService.getModel(area, bytes("entity1"),
DBTestEntity.class).getType());
        Assert.assertEquals(22222, dbService.getModel(area, bytes("entity2"),
DBTestEntity.class).getType());
        Assert.assertEquals(33333, dbService.getModel(area, bytes("entity3"),
DBTestEntity.class).getType());
        Assert.assertNotNull(dbService.get(area, bytes("entity3")));
        Assert.assertNull(dbService.get(area, bytes("entity4")));
        Assert.assertNull(dbService.get(area, bytes("entity5")));
        LevelDBManager.destroyArea(area);
```

```
    }


    public void snapshotOfAddress() throws Exception{
        Map<String, Na> balanceMap = new HashMap<>();
        List<byte[]> valueList = valueList("ledger_utxo");
        Coin coin;
        String strAddress;
        Na balance;
        Address address;
        byte[] hash160;
        for (byte[] bytes : valueList) {
            coin = new Coin();
            coin.parse(bytes,0);
            //
            hash160 = new byte[20];
            //System.arraycopy(coin.getOwner(), 2, hash160, 0, 20);
            System.arraycopy(coin.getAddress(), 2, hash160, 0, 20);
            address = new Address(NulsContext.DEFAULT_CHAIN_ID,
NulsContext.DEFAULT_ADDRESS_TYPE, hash160);
            strAddress = address.toString();
            balance = balanceMap.get(strAddress);
            if(balance == null) {
                balance = Na.ZERO;
            }
            balance = balance.add(coin.getNa());
            balanceMap.put(strAddress, balance);
        }
        Set<Map.Entry<String, Na>> entries = balanceMap.entrySet();
        //for (Map.Entry<String, Na> entry : entries) {
        //    System.out.println(entry.getKey() + "," + entry.getValue().toText());
        //}

        List<Map.Entry<String, Na>> list = new ArrayList<>();
        list.addAll(entries);
        list.sort(new Comparator<Map.Entry<String, Na>>() {
            @Override
            public int compare(Map.Entry<String, Na> o1, Map.Entry<String, Na> o2) {
                return o2.getValue().compareTo(o1.getValue());
            }
        });
        for (Map.Entry<String, Na> entry : list) {
```

```
            System.out.println(entry.getKey() + " " + entry.getValue().toString());
        }
    }

    @Test
    public void test() throws UnsupportedEncodingException {
        testPutModel();
        testGetModel();
        testGetModelByClass();
        testPut_1();
        testPut_2();
        testPut_3();
        testGet();
        testDelete();
        testListArea();
        testFullCreateArea();
        testDestroyArea();
        testKeySet();
        testKeyList();
        testComparator();
        testCacheSize();
        testEntrySet();
        testEntryList();
        testEntryListByClass();
        testValuesByClass();
        testValueList();
    }

    public void testFullCreateArea() {
        for (int i = 0, length = getMax() + 10; i < length; i++) {
            createArea(area + "-" + i);
        }
        Assert.assertEquals(getMax(), listArea().length);
    }

    public void testDestroyArea() {
        for (int i = 0, length = getMax() + 10; i < length; i++) {
            destroyArea(area + "-" + i);
        }
        Assert.assertTrue(listArea().length < getMax());
    }
```

```
    public void testPut_1() throws UnsupportedEncodingException {
        String value = "testvalue_1";
        put(area, key.getBytes(NulsConfig.DEFAULT_ENCODING),
value.getBytes(NulsConfig.DEFAULT_ENCODING));
        String getValue = new String(get(area, bytes(key)), NulsConfig.DEFAULT_ENCODING);
        Assert.assertEquals(value, getValue);
    }

    public void testPut_2() throws UnsupportedEncodingException {
        String value = "testvalue_2";
        put(area, bytes(key), bytes(value));
        String getValue = new String(get(area, bytes(key)), NulsConfig.DEFAULT_ENCODING);
        Assert.assertEquals(value, getValue);
    }

    public void testPut_3() throws UnsupportedEncodingException {
        String value = "testvalue_3";
        put(area, bytes(key), bytes(value));
        String getValue = new String(get(area, bytes(key)), NulsConfig.DEFAULT_ENCODING);
        Assert.assertEquals(value, getValue);
    }

    public void testGet() throws UnsupportedEncodingException {
        String value = "testvalue_3";
        String getValue = new String(get(area, bytes(key)), NulsConfig.DEFAULT_ENCODING);
        Assert.assertEquals(value, getValue);
    }

    public void testDelete() throws UnsupportedEncodingException {
        delete(area, bytes(key));
        Assert.assertNull(get(area, bytes(key)));
    }

    public void testListArea() throws UnsupportedEncodingException {
        String[] areas = listArea();
        if (areas.length < getMax()) {
            String testArea = "testListArea";
            createArea(testArea);
            areas = listArea();
            boolean exist = false;
            for (String area : areas) {
                if (area.equals(testArea)) {
```

```
                exist = true;
                break;
            }
        }
        Assert.assertTrue("create - list areas failed.", exist);
        put(testArea, bytes(key), bytes("testListArea"));
        String getValue = new String(get(testArea, bytes(key)),
NulsConfig.DEFAULT_ENCODING);
        Assert.assertEquals("testListArea", getValue);
        destroyArea(testArea);
    }
}

public void testPutModel() {
    DBTestEntity entity = new DBTestEntity();
    putModel(area, bytes(key), entity);
    Object object = getModel(area, bytes(key));
    Assert.assertEquals(entity.getClass().getName(), object.getClass().getName());
}

public void testGetModel() {
    Object object = getModel(area, bytes(key));
    Assert.assertEquals(DBTestEntity.class.getName(), object.getClass().getName());
}

public void testGetModelByClass() {
    DBTestEntity object = getModel(area, bytes(key), DBTestEntity.class);
    Assert.assertEquals(DBTestEntity.class.getName(), object.getClass().getName());
}

public void testKeySet() {
    String area = "testKeySet";
    createArea(area);
    put(area, bytes("set1"), bytes("set1value"));
    put(area, bytes("set2"), bytes("set2value"));
    put(area, bytes("set3"), bytes("set3value"));
    Set<byte[]> keys = keySet(area);
    Set<String> keysStr = new HashSet<>();
    for(byte[] bytes : keys) {
        keysStr.add(asString(bytes));
    }
    Assert.assertEquals(3, keys.size());
```

```
    Assert.assertTrue(keysStr.contains("set1"));
    Assert.assertTrue(keysStr.contains("set2"));
    Assert.assertTrue(keysStr.contains("set3"));
    destroyArea(area);
}

public void testKeyList() {
    String area = "testKeyList";
    createArea(area);
    put(area, bytes("set05"), bytes("set05value"));
    put(area, bytes("set06"), bytes("set06value"));
    put(area, bytes("set02"), bytes("set02value"));
    put(area, bytes("set01"), bytes("set01value"));
    put(area, bytes("set04"), bytes("set04value"));
    put(area, bytes("set03"), bytes("set03value"));
    List<byte[]> keys = keyList(area);
    Assert.assertEquals(6, keys.size());
    int i = 0;
    for (byte[] key : keys) {
        Assert.assertEquals("set0" + (++i), asString(key));
    }
    destroyArea(area);
}

public void testComparator() {
    String area = "testComparator";
    destroyArea(area);
    createArea(area, new Comparator<byte[]>() {
        @Override
        public int compare(byte[] o1, byte[] o2) {
            String s1 = asString(o1);
            String s2 = asString(o2);
            if ("set03".equals(s1)) {
                return 1;
            }
            if("set03".equals(s2)) {
                return -1;
            }
            return s1.compareTo(s2);
        }
    });
    put(area, bytes("set05"), bytes("set05value"));
```

```
        put(area, bytes("set06"), bytes("set06value"));
        put(area, bytes("set02"), bytes("set02value"));
        put(area, bytes("set01"), bytes("set01value"));
        put(area, bytes("set04"), bytes("set04value"));
        put(area, bytes("set03"), bytes("set03value"));
        List<byte[]> keys = keyList(area);
        Assert.assertEquals(6, keys.size());
        String contactAllKeys = "";
        for (byte[] key : keys) {
            contactAllKeys += asString(key);
        }
        System.out.println(contactAllKeys);
        Assert.assertEquals("set01set02set04set05set06set03", contactAllKeys);

        Comparator comparator = getModel(getBaseAreaName(), bytes(area + "-comparator"),
Comparator.class);
        Long cacheSize = getModel(getBaseAreaName(), bytes(area + "-cacheSize"), Long.class);
        destroyArea(area);
        Assert.assertTrue(createArea(area, cacheSize, comparator).isSuccess());
        put(area, bytes("set05"), bytes("set05value"));
        put(area, bytes("set06"), bytes("set06value"));
        put(area, bytes("set02"), bytes("set02value"));
        put(area, bytes("set01"), bytes("set01value"));
        put(area, bytes("set04"), bytes("set04value"));
        put(area, bytes("set03"), bytes("set03value"));
        keys = keyList(area);
        Assert.assertEquals(6, keys.size());
        contactAllKeys = "";
        for (byte[] key : keys) {
            contactAllKeys += asString(key);
        }
        System.out.println(contactAllKeys);
        Assert.assertEquals("set01set02set04set05set06set03", contactAllKeys);
        destroyArea(area);
    }


    public void testCacheSize() {
        String area = "testCacheSize";
        createArea(area, 100 * 1024 * 1024l, new Comparator<byte[]>() {
            @Override
            public int compare(byte[] o1, byte[] o2) {
```

```
            return asString(o1).compareTo(asString(o2));
        }
    });
    put(area, bytes("set5"), bytes("set5value"));
    put(area, bytes("set6"), bytes("set6value"));
    put(area, bytes("set2"), bytes("set2value"));
    put(area, bytes("set1"), bytes("set1value"));
    put(area, bytes("set4"), bytes("set4value"));
    put(area, bytes("set3"), bytes("set3value"));
    List<Entry<byte[], byte[]>> entries = entryList(area);
    Assert.assertEquals(6, entries.size());

    int i = 1;
    for (Entry<byte[], byte[]> entry : entries) {
        Assert.assertEquals("set" + i, asString(entry.getKey()));
        Assert.assertEquals("set" + i + "value", asString(entry.getValue()));
        i++;
    }
    closeArea(area);
    Comparator comparator = getModel(getBaseAreaName(), bytes(area + "-comparator"),
Comparator.class);
    Long cacheSize = getModel(getBaseAreaName(), bytes(area + "-cacheSize"), Long.class);
    Assert.assertTrue(createArea(area, cacheSize, comparator).isSuccess());
    destroyArea(area);
  }

  public void testEntrySet() {
    String area = "testEntrySet";
    createArea(area);
    put(area, bytes("set1"), bytes("set1value"));
    put(area, bytes("set2"), bytes("set2value"));
    put(area, bytes("set3"), bytes("set3value"));
    Set<Entry<byte[], byte[]>> entries = entrySet(area);
    Assert.assertEquals(3, entries.size());

    for (Entry<byte[], byte[]> entry : entries) {
        Assert.assertEquals(Iq80DBFactory.asString(entry.getValue()), asString(entry.getKey()) +
"value");
Assert.assertTrue(Iq80DBFactory.asString(entry.getValue()).startsWith(asString(entry.getKey())));
     }
    destroyArea(area);
  }
```

```
    public void testEntryList() {
        String area = "testEntryList";
        createArea(area, new Comparator<byte[]>() {
            @Override
            public int compare(byte[] o1, byte[] o2) {
                String s1 = asString(o1);
                String s2 = asString(o2);
                if ("set3".equals(s1)) {
                    return 1;
                }
                if("set3".equals(s2)) {
                    return -1;
                }
                return s1.compareTo(s2);
            }
        });
        put(area, bytes("set5"), bytes("set5value"));
        put(area, bytes("set6"), bytes("set6value"));
        put(area, bytes("set2"), bytes("set2value"));
        put(area, bytes("set1"), bytes("set1value"));
        put(area, bytes("set4"), bytes("set4value"));
        put(area, bytes("set3"), bytes("set3value"));
        List<Entry<byte[], byte[]>> entries = entryList(area);
        Assert.assertEquals(6, entries.size());

        String contact = "";
        for (Entry<byte[], byte[]> entry : entries) {
            System.out.println(asString(entry.getKey()) + "=" + asString(entry.getValue()));
            contact += asString(entry.getKey()) + asString(entry.getValue());
        }
Assert.assertEquals("set1set1valueset2set2valueset4set4valueset5set5valueset6set6valueset3se
t3value", contact);
        destroyArea(area);
    }

    public void testEntryListByClass() {
        String area = "testEntryListByClass";
        createArea(area);
        DBTestEntity entity = new DBTestEntity();
        putModel(area, bytes("entity1"), entity);
        entity = new DBTestEntity();
```

```
    putModel(area, bytes("entity2"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity3"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity4"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity5"), entity);
    List<Entry<byte[], DBTestEntity>> list = entryList(area, DBTestEntity.class);
    Assert.assertEquals(5, list.size());
    destroyArea(area);
}

public void testValuesByClass() {
    String area = "testValuesByClass";
    createArea(area);
    DBTestEntity entity = new DBTestEntity();
    putModel(area, bytes("entity1"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity2"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity3"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity4"), entity);
    entity = new DBTestEntity();
    putModel(area, bytes("entity5"), entity);
    List<DBTestEntity> list = values(area, DBTestEntity.class);
    Assert.assertEquals(5, list.size());
    destroyArea(area);
}

public void testValueList() {
    String area = "testValueList";
    createArea(area, new Comparator<byte[]>() {
        @Override
        public int compare(byte[] o1, byte[] o2) {
            String s1 = asString(o1);
            String s2 = asString(o2);
            if ("set3".equals(s1)) {
                return 1;
            }
            if("set3".equals(s2)) {
                return -1;
```

```
            }
            return s1.compareTo(s2);
        }
    });
    put(area, bytes("set5"), bytes("set5value"));
    put(area, bytes("set6"), bytes("set6value"));
    put(area, bytes("set2"), bytes("set2value"));
    put(area, bytes("set1"), bytes("set1value"));
    put(area, bytes("set4"), bytes("set4value"));
    put(area, bytes("set3"), bytes("set3value"));
    List<byte[]> list = valueList(area);
    Assert.assertEquals(6, list.size());

    String contact = "";
    for (byte[] value : list) {
        System.out.println(asString(value));
        contact += asString(value);
    }
    Assert.assertEquals("set1valueset2valueset4valueset5valueset6valueset3value", contact);
    destroyArea(area);

    createArea(area);
    put(area, bytes("set5"), bytes("set5value"));
    put(area, bytes("set6"), bytes("set6value"));
    put(area, bytes("set2"), bytes("set2value"));
    put(area, bytes("set1"), bytes("set1value"));
    put(area, bytes("set4"), bytes("set4value"));
    put(area, bytes("set3"), bytes("set3value"));
    list = valueList(area);
    Assert.assertEquals(6, list.size());

    contact = "";
    for (byte[] value : list) {
        System.out.println(asString(value));
        contact += asString(value);
    }
    Assert.assertEquals("set1valueset2valueset3valueset4valueset5valueset6value", contact);
    destroyArea(area);
}

@AfterClass
public static void after() {
```

```
        close();
    }
}
```