数字逻辑形势做RLP数据编码的源码解析

一、RLP的源码不是很多，主要分了三个文件，这些文件位于根目录的/rlp/下

```
decode.go           解码器，把RLP数据解码为go的数据结构
decode_tail_test.go     解码器测试代码
decode_test.go          解码器测试代码
doc.go              文档代码
encode.go           编码器，把GO的数据结构序列化为字节数组
encode_test.go          编码器测试
encode_example_test.go
raw.go              未解码的RLP数据
raw_test.go
typecache.go            类型缓存，  类型缓存记录了类型->(编码器|解码器)的内容。
```

二、如何根据类型找到对应的编码器和解码器 typecache.go

在C++或者Java等支持重载的语言中， 可以通过不同的类型重载同一个函数名称来实现方法针对不同类型的分派,比如， 也可以通过泛型来实现函数的分派。

```
string encode(int);
string encode(long);
string encode(struct test*)
```

但是GO语言本身不支持重载， 也没有泛型，所以函数的分派就需要自己实现了。 typecache.go主要是实现这个目的， 通过自身的类型来快速的找到自己的编码器函数和解码器函数。

我们首先看看核心数据结构

```
var (
    typeCacheMutex sync.RWMutex                 //读写锁，用来在多线程的时候保护typeCache这个Map
    typeCache      = make(map[typekey]*typeinfo) //核心数据结构，保存了类型->编解码器函数
)
type typeinfo struct { //存储了编码器和解码器函数
    decoder
    writer
```

```
}
type typekey struct {
    reflect.Type
    // the key must include the struct tags because they
    // might generate a different decoder.
    tags
}
```

可以看到核心数据结构就是typeCache这个Map， Map的key是类型，value是对应的编码和解码器。

下面是用户如何获取编码器和解码器的函数

```
func cachedTypeInfo(typ reflect.Type, tags tags) (*typeinfo, error) {
    typeCacheMutex.RLock()      //加读锁来保护，
    info := typeCache[typekey{typ, tags}]
    typeCacheMutex.RUnlock()
    if info != nil { //如果成功获取到信息，那么就返回
        return info, nil
    }
    // not in the cache, need to generate info for this type.
    typeCacheMutex.Lock()  //否则加写锁 调用cachedTypeInfo1函数创建并返回， 这里需要注意的是在多线程环境下有可能多个线程同时调用到这个地方，所以当你进入cachedTypeInfo1方法的时候需要判断一下是否已经被别的线程先创建成功了。
    defer typeCacheMutex.Unlock()
    return cachedTypeInfo1(typ, tags)
}

func cachedTypeInfo1(typ reflect.Type, tags tags) (*typeinfo, error) {
    key := typekey{typ, tags}
    info := typeCache[key]
    if info != nil {
        // 其他的线程可能已经创建成功了， 那么我们直接获取到信息然后返回
        return info, nil
    }
    // put a dummmy value into the cache before generating.
    // if the generator tries to lookup itself, it will get
    // the dummy value and won't call itself recursively.
    //这个地方首先创建了一个值来填充这个类型的位置，避免遇到一些递归定义的数据类型形成死循环
    typeCache[key] = new(typeinfo)
    info, err := genTypeInfo(typ, tags)
    if err != nil {
        // remove the dummy value if the generator fails
        delete(typeCache, key)
        return nil, err
    }
```

```
    *typeCache[key] = *info
    return typeCache[key], err
}
```

genTypeInfo是生成对应类型的编解码器函数。

```
func genTypeInfo(typ reflect.Type, tags tags) (info *typeinfo, err error) {
    info = new(typeinfo)
    if info.decoder, err = makeDecoder(typ, tags); err != nil {
        return nil, err
    }
    if info.writer, err = makeWriter(typ, tags); err != nil {
        return nil, err
    }
    return info, nil
}
```

makeDecoder的处理逻辑和makeWriter的处理逻辑大致差不多， 这里我就只贴出makeWriter的处理逻辑，

```
// makeWriter creates a writer function for the given type.
func makeWriter(typ reflect.Type, ts tags) (writer, error) {
    kind := typ.Kind()
    switch {
    case typ == rawValueType:
        return writeRawValue, nil
    case typ.Implements(encoderInterface):
        return writeEncoder, nil
    case kind != reflect.Ptr && reflect.PtrTo(typ).Implements(encoderInterface):
        return writeEncoderNoPtr, nil
    case kind == reflect.Interface:
        return writeInterface, nil
    case typ.AssignableTo(reflect.PtrTo(bigInt)):
        return writeBigIntPtr, nil
    case typ.AssignableTo(bigInt):
        return writeBigIntNoPtr, nil
    case isUint(kind):
        return writeUint, nil
    case kind == reflect.Bool:
        return writeBool, nil
    case kind == reflect.String:
        return writeString, nil
    case kind == reflect.Slice && isByte(typ.Elem()):
        return writeBytes, nil
    case kind == reflect.Array && isByte(typ.Elem()):
```

```
        return writeByteArray, nil
    case kind == reflect.Slice || kind == reflect.Array:
        return makeSliceWriter(typ, ts)
    case kind == reflect.Struct:
        return makeStructWriter(typ)
    case kind == reflect.Ptr:
        return makePtrWriter(typ)
    default:
        return nil, fmt.Errorf("rlp: type %v is not RLP-serializable", typ)
    }
}
```

可以看到就是一个switch case,根据类型来分配不同的处理函数。 这个处理逻辑还是很简单的。针对简单类型很简单，根据黄皮书上面的描述来处理即可。 不过对于结构体类型的处理还是挺有意思的，而且这部分详细的处理逻辑在黄皮书上面也是找不到的。

```
type field struct {
    index int
    info  *typeinfo
}
func makeStructWriter(typ reflect.Type) (writer, error) {
    fields, err := structFields(typ)
    if err != nil {
        return nil, err
    }
    writer := func(val reflect.Value, w *encbuf) error {
        lh := w.list()
        for _, f := range fields {
            //f是field结构， f.info是typeinfo的指针， 所以这里其实是调用字段的编码器方法
。
            if err := f.info.writer(val.Field(f.index), w); err != nil {
                return err
            }
        }
        w.listEnd(lh)
        return nil
    }
    return writer, nil
}
```

这个函数定义了结构体的编码方式， 通过structFields方法得到了所有的字段的编码器， 然后返回一个方法，这个方法遍历所有的字段，每个字段调用其编码器方法。

```
func structFields(typ reflect.Type) (fields []field, err error) {
    for i := 0; i < typ.NumField(); i++ {
```

```
        if f := typ.Field(i); f.PkgPath == "" { // exported
            tags, err := parseStructTag(typ, i)
            if err != nil {
                return nil, err
            }
            if tags.ignored {
                continue
            }
            info, err := cachedTypeInfo1(f.Type, tags)
            if err != nil {
                return nil, err
            }
            fields = append(fields, field{i, info})
        }
    }
    return fields, nil
}
```

structFields函数遍历所有的字段，然后针对每一个字段调用cachedTypeInfo1。 可以看到这是一个递归的调用过程。 上面的代码中有一个需要注意的是f.PkgPath == "" 这个判断针对的是所有导出的字段， 所谓的导出的字段就是说以大写字母开头命令的字段。

三、编码器 encode.go

首先定义了空字符串和空List的值，分别是 0x80和0xC0。 注意，整形的0值的对应值也是0x80。这个在黄皮书上面是没有看到有定义的。 然后定义了一个接口类型给别的类型实现EncodeRLP

```
var (
    // Common encoded values.
    // These are useful when implementing EncodeRLP.
    EmptyString = []byte{0x80}
    EmptyList   = []byte{0xC0}
)

// Encoder is implemented by types that require custom
// encoding rules or want to encode private fields.
type Encoder interface {
    // EncodeRLP should write the RLP encoding of its receiver to w.
    // If the implementation is a pointer method, it may also be
    // called for nil pointers.
    //
    // Implementations should generate valid RLP. The data written is
    // not verified at the moment, but a future version might. It is
    // recommended to write only a single value but writing multiple
```

```
    // values or no value at all is also permitted.
    EncodeRLP(io.Writer) error
}
```

然后定义了一个最重要的方法， 大部分的EncodeRLP方法都是直接调用了这个方法Encode方法。这个方法首先获取了一个encbuf对象。 然后调用这个对象的encode方法。encode方法中，首先获取了对象的反射类型，根据反射类型获取它的编码器，然后调用编码器的writer方法。 这个就跟上面谈到的typecache联系到一起了。

```
func Encode(w io.Writer, val interface{}) error {
    if outer, ok := w.(*encbuf); ok {
        // Encode was called by some type's EncodeRLP.
        // Avoid copying by writing to the outer encbuf directly.
        return outer.encode(val)
    }
    eb := encbufPool.Get().(*encbuf)
    defer encbufPool.Put(eb)
    eb.reset()
    if err := eb.encode(val); err != nil {
        return err
    }
    return eb.toWriter(w)
}
func (w *encbuf) encode(val interface{}) error {
    rval := reflect.ValueOf(val)
    ti, err := cachedTypeInfo(rval.Type(), tags{})
    if err != nil {
        return err
    }
    return ti.writer(rval, w)
}
```

encbuf的介绍

encbuf是encode buffer的简写(我猜的)。encbuf出现在Encode方法，和很多Writer方法中。顾名思义，这个是在encode的过程中充当buffer的作用。下面先看看encbuf的定义。

```
type encbuf struct {
    str     []byte      // string data, contains everything except list headers
    lheads  []*listhead // all list headers
    lhsize  int         // sum of sizes of all encoded list headers
    sizebuf []byte      // 9-byte auxiliary buffer for uint encoding
}
```

```
type listhead struct {
    offset int // index of this header in string data
    size   int // total size of encoded data (including list headers)
}
```

从注释可以看到， str字段包含了所有的内容，除了列表的头部。 列表的头部记录在lheads字段中。 lhsize字段记录了lheads的长度， sizebuf是9个字节大小的辅助buffer，专门用来处理uint的编码的。 listhead由两个字段组成， offset字段记录了列表数据在str字段的哪个位置， size字段记录了包含列表头的编码后的数据的总长度。

对于普通的类型，比如字符串，整形，bool型等数据，就是直接往str字段里面填充就行了。 但是对于结构体类型的处理， 就需要特殊的处理方式了。可以看看上面提到过的makeStructWriter方法。

```
func makeStructWriter(typ reflect.Type) (writer, error) {
        fields, err := structFields(typ)
        ...
        writer := func(val reflect.Value, w *encbuf) error {
            lh := w.list()
            for _, f := range fields {
                if err := f.info.writer(val.Field(f.index), w); err != nil {
                    return err
                }
            }
            w.listEnd(lh)
        }
    }
```

可以看到上面的代码中体现了处理结构体数据的特殊处理方法，就是首先调用w.list()方法，处理完毕之后再调用listEnd(lh)方法。 采用这种方式的原因是我们在刚开始处理结构体的时候，并不知道处理后的结构体的长度有多长，因为需要根据结构体的长度来决定头的处理方式(回忆一下黄皮书里面结构体的处理方式)，所以我们在处理前记录好str的位置，然后开始处理每个字段，处理完之后在看一下str的数据增加了多少就知道处理后的结构体长度有多长了。

```
func (w *encbuf) list() *listhead {
    lh := &listhead{offset: len(w.str), size: w.lhsize}
    w.lheads = append(w.lheads, lh)
    return lh
}

func (w *encbuf) listEnd(lh *listhead) {
    lh.size = w.size() - lh.offset - lh.size    //lh.size记录了list开始的时候的队列头应该占用的长度 w.size()返回的是str的长度加上lhsize
```

```
    if lh.size < 56 {
        w.lhsize += 1 // length encoded into kind tag
    } else {
        w.lhsize += 1 + intsize(uint64(lh.size))
    }
}
func (w *encbuf) size() int {
    return len(w.str) + w.lhsize
}
```

然后我们可以看看encbuf最后的处理逻辑，会对listhead进行处理，组装成完整的RLP数据

```
func (w *encbuf) toBytes() []byte {
    out := make([]byte, w.size())
    strpos := 0
    pos := 0
    for _, head := range w.lheads {
        // write string data before header
        n := copy(out[pos:], w.str[strpos:head.offset])
        pos += n
        strpos += n
        // write the header
        enc := head.encode(out[pos:])
        pos += len(enc)
    }
    // copy string data after the last list header
    copy(out[pos:], w.str[strpos:])
    return out
}
```

writer介绍

剩下的流程其实比较简单了。 就是根据黄皮书针把每种不同的数据填充到encbuf里面去。

```
func writeBool(val reflect.Value, w *encbuf) error {
    if val.Bool() {
        w.str = append(w.str, 0x01)
    } else {
        w.str = append(w.str, 0x80)
    }
    return nil
}
func writeString(val reflect.Value, w *encbuf) error {
    s := val.String()
    if len(s) == 1 && s[0] <= 0x7f {
```

```
        // fits single byte, no string header
        w.str = append(w.str, s[0])
    } else {
        w.encodeStringHeader(len(s))
        w.str = append(w.str, s...)
    }
    return nil
}
```

四、解码器 decode.go

解码器的大致流程和编码器差不多，理解了编码器的大致流程，也就知道了解码器的大致流程。

```
func (s *Stream) Decode(val interface{}) error {
    if val == nil {
        return errDecodeIntoNil
    }
    rval := reflect.ValueOf(val)
    rtyp := rval.Type()
    if rtyp.Kind() != reflect.Ptr {
        return errNoPointer
    }
    if rval.IsNil() {
        return errDecodeIntoNil
    }
    info, err := cachedTypeInfo(rtyp.Elem(), tags{})
    if err != nil {
        return err
    }
    err = info.decoder(s, rval.Elem())
    if decErr, ok := err.(*decodeError); ok && len(decErr.ctx) > 0 {
        // add decode target type to error so context has more meaning
        decErr.ctx = append(decErr.ctx, fmt.Sprint("(", rtyp.Elem(), ")"))
    }
    return err
}

func makeDecoder(typ reflect.Type, tags tags) (dec decoder, err error) {
    kind := typ.Kind()
    switch {
    case typ == rawValueType:
        return decodeRawValue, nil
    case typ.Implements(decoderInterface):
        return decodeDecoder, nil
    case kind != reflect.Ptr && reflect.PtrTo(typ).Implements(decoderInterface):
        return decodeDecoderNoPtr, nil
```

```
    case typ.AssignableTo(reflect.PtrTo(bigInt)):
        return decodeBigInt, nil
    case typ.AssignableTo(bigInt):
        return decodeBigIntNoPtr, nil
    case isUint(kind):
        return decodeUint, nil
    case kind == reflect.Bool:
        return decodeBool, nil
    case kind == reflect.String:
        return decodeString, nil
    case kind == reflect.Slice || kind == reflect.Array:
        return makeListDecoder(typ, tags)
    case kind == reflect.Struct:
        return makeStructDecoder(typ)
    case kind == reflect.Ptr:
        if tags.nilOK {
            return makeOptionalPtrDecoder(typ)
        }
        return makePtrDecoder(typ)
    case kind == reflect.Interface:
        return decodeInterface, nil
    default:
        return nil, fmt.Errorf("rlp: type %v is not RLP-serializable", typ)
    }
}
```

我们同样通过结构体类型的解码过程来查看具体的解码过程。跟编码过程差不多，首先通过structFields获取需要解码的所有字段，然后每个字段进行解码。 跟编码过程差不多有一个List()和ListEnd()的操作，不过这里的处理流程和编码过程不一样，后续章节会详细介绍。

```
func makeStructDecoder(typ reflect.Type) (decoder, error) {
    fields, err := structFields(typ)
    if err != nil {
        return nil, err
    }
    dec := func(s *Stream, val reflect.Value) (err error) {
        if _, err := s.List(); err != nil {
            return wrapStreamError(err, typ)
        }
        for _, f := range fields {
            err := f.info.decoder(s, val.Field(f.index))
            if err == EOL {
                return &decodeError{msg: "too few elements", typ: typ}
            } else if err != nil {
                return addErrorContext(err, "."+typ.Field(f.index).Name)
            }
```

```
        }
        return wrapStreamError(s.ListEnd(), typ)
    }
    return dec, nil
}
```

下面在看字符串的解码过程，因为不同长度的字符串有不同方式的编码，我们可以通过前缀的不同来获取字符串的类型， 这里我们通过s.Kind()方法获取当前需要解析的类型和长度，如果是Byte类型，那么直接返回Byte的值， 如果是String类型那么读取指定长度的值然后返回。 这就是kind()方法的用途。

```
func (s *Stream) Bytes() ([]byte, error) {
    kind, size, err := s.Kind()
    if err != nil {
        return nil, err
    }
    switch kind {
    case Byte:
        s.kind = -1 // rearm Kind
        return []byte{s.byteval}, nil
    case String:
        b := make([]byte, size)
        if err = s.readFull(b); err != nil {
            return nil, err
        }
        if size == 1 && b[0] < 128 {
            return nil, ErrCanonSize
        }
        return b, nil
    default:
        return nil, ErrExpectedString
    }
}
```

Stream 结构分析

解码器的其他代码和编码器的结构差不多， 但是有一个特殊的结构是编码器里面没有的。那就是Stream。
这个是用来读取用流式的方式来解码RLP的一个辅助类。 前面我们讲到了大致的解码流程就是首先通过Kind()方法获取需要解码的对象的类型和长度,然后根据长度和类型进行数据的解码。 那么我们如何处理结构体的字段又是结构体的数据呢， 回忆我们对结构体进行处理的时候，首先调用s.List()方法，然后对每个字段进行解码，最后调用s.EndList()方法。 技巧就在这两个方法里面，下面我们看看这两个方法。

```
type Stream struct {
    r ByteReader
    // number of bytes remaining to be read from r.
    remaining uint64
    limited   bool
    // auxiliary buffer for integer decoding
    uintbuf []byte
    kind    Kind   // kind of value ahead
    size    uint64 // size of value ahead
    byteval byte   // value of single byte in type tag
    kinderr error  // error from last readKind
    stack   []listpos
}
type listpos struct{ pos, size uint64 }
```

Stream的List方法，当调用List方法的时候。我们先调用Kind方法获取类型和长度，如果类型不匹配那么就抛出错误，然后我们把一个listpos对象压入到堆栈，这个对象是关键。 这个对象的pos字段记录了当前这个list已经读取了多少字节的数据， 所以刚开始的时候肯定是0. size字段记录了这个list对象一共需要读取多少字节数据。这样我在处理后续的每一个字段的时候，每读取一些字节，就会增加pos这个字段的值，处理到最后会对比pos字段和size字段是否相等，如果不相等，那么会抛出异常。

```
func (s *Stream) List() (size uint64, err error) {
    kind, size, err := s.Kind()
    if err != nil {
        return 0, err
    }
    if kind != List {
        return 0, ErrExpectedList
    }
    s.stack = append(s.stack, listpos{0, size})
    s.kind = -1
    s.size = 0
    return size, nil
}
```

Stream的ListEnd方法，如果当前读取的数据数量pos不等于声明的数据长度size，抛出异常，然后对堆栈进行pop操作，如果当前堆栈不为空，那么就在堆栈的栈顶的pos加上当前处理完毕的数据长度(用来处理这种情况--结构体的字段又是结构体， 这种递归的结构)

```
func (s *Stream) ListEnd() error {
    if len(s.stack) == 0 {
        return errNotInList
```

```
    }
    tos := s.stack[len(s.stack)-1]
    if tos.pos != tos.size {
        return errNotAtEOL
    }
    s.stack = s.stack[:len(s.stack)-1] // pop
    if len(s.stack) > 0 {
        s.stack[len(s.stack)-1].pos += tos.size
    }
    s.kind = -1
    s.size = 0
    return nil
}
```