以太坊源码解析：rlp

2018-12-23 fatcat22 ethereum 原创 ethereum rlp 源码解析

> 本篇文章分析的源码地址为：https://github.com/ethereum/go-ethereum
> 分支：master
> commit id: 257bfff316e4efb8952fbeb67c91f86af579cb0a

引言

以太坊是区块链项目中最为知名的项目之一，这里就不多做介绍了。rlp（Recursive Length Prefix）是以太坊中一个模块，其功能就是对对象进行序列化和反序列化（实际上根据官方的介绍，rlp唯一的目的是解决结构体的编码问题）。我们在这篇文章中将对rlp进行详细说明。

rlp作为一个序列化、反序列化的模块，自然有一套自己的编码规则，官方介绍文档在这里。这篇文章里我们先介绍编码规则，再对源代码进行整体梳理和重点说明。

rlp编码规则

定义

在rlp编码中，有两种结构：string和list。注意这两个结构的含义与我们通常的理解略有差别，它们的定义如下：

> string指的是一串字节，如Go中的[]byte
> list:由string或list组成的一个项目列表

比如，”\x11\x22\x33”和”abc”都是一个string；[“abc”, [“def”], [“d”, “g”]]是一个list。可以看出list是一个嵌套结构。

伪代码

下面我们使用伪代码直接给出rlp编码的规则。首先是针对string的编码规则：

```
if string长度 == 1 && 0 <= string[0] <= 0x7f {
  //比如'\x30', 编码为:0x30
  code = string[0]
} else if 0 <= string长度 <= 55 {
  //比如"\xab"编码为:0x81 0xab
  //"\x30\x40" 编码为: 0x82 0x30 0x40
  //"" 编码为: 0x80
  code = 0x80+string长度, string
} else if 56 <= string长度 <= 0xffffffff {
  //NonZeroPrefixBigEndianBytes 将一个整数以大端的方式编码成字节序列，并去掉前面值为0的字节。比如
uint32(0x12c)以大端编码为0x00 0x00 0x01 0x2c，然后去掉前面两个字节的0值，即为0x01 0x2c.
  //比如 "\x01\x02...\x30"，假设这个串的长度为300字节。编码为：0xba, 0x01, 0x2c, 0x01, 0x02, ...., 0x30
  code = 0xB8 + sizeof(NonZeroPrefixBigEndianBytes(string长度)), NonZeroPrefixBigEndianBytes(string
长度), string
} else {
  invalid
}
```

上面的伪代码已经清晰的表达了对于string的编码方法。有两个地方需要稍加注意。一是string长度为0的情况是被第二个if覆盖的，注释中的例子也列举了这种情况；二是string长度>=56时，对长度的编码有两个注意点：大端和去除前面所有0字节，这些都在注释的示例中有说明。

Content

下面再来看看针对list的编码规则：

```
//list长度 是指list中的所有项目使用rlp编码以后的长度的总和
//list数据 是指list各项使用rlp编码以后的连接起起来的数据
if 0 <= list长度 <= 55 {
  //比如["\x30", "\x40\x50"] 编码为: 0xc4, 0x30, 0x82, 0x40, 0x50
  //[]编码为: 0xc0
  code = 0xC0 + list长度, list数据
} else if 56 <= list长度 <= 0xffffffff {
  //NonZeroPrefixBigEndianBytes的意义同string的伪代码
  //比如["\x01", "\x02", ..., "\x30"], 假设这个list的所有元素经rlp编码以后长度为0x12c字节, 则这个list编码为: 0xfa, 0x01, 0x2c, list数据...
  code = 0xF8 + sizeof(NonZeroPrefixBigEndianBytes(list长度)), NonZeroPrefixBigEndianBytes(string长度), list数据
} else {
  invalid
}
```

可以看出对于list的编码与对于string编码的机制是类似的。另外无论是string还是list都存在一个特殊的值作为一个标识：0x80和0xC0。这两个值是经过精心选取的，下面我们对这两个值的选取作进一步的讨论。

标识值的选取

总体而言，无论对于string还是list，rlp的思路就是对于某个值域范围内的单个字节可以直接编码；对于某个长度范围之内的字节序列，采用简单方式，即标识值+字节长度、后接字节数据的方式编码；而对于超过这个长度范围的字节序列，需要补充一些数据据来记录字节序列到底有多长，所以采用复杂方式，即标识值+字节长度的字节数、后接字节长度大端编码后去0、后接字节序列的方式编码。

顺着这个总结进行思考，就会发现这里有几个问题需要进一步解决，为了后面描述方便，我们顺便为其定义一个名字：

- 可以直接编码的 **值域范围** ，我们将其命名为 **Single**
- 区分使用简单方式还是复杂方式编码的 **长度范围** 是多少。我们将其命名为 **Threshold**
- 各种情况下 **标识值** ，我们将其命名为 **FlagByte**，string的FlagByte我们命名为 **StringFlagByte** ，list的FlagByte我们命名为 **ListFlagByte**

在rlp已成型的代码里，list类型的FlagByte值是较大的（0xC0），所以我们这里也应用这种情况进行分析（其实无论是list的FlagByte较大还是string的FlagByte较大是一样的）。

一方面，在上面已有的思路框架下，第一个字节，即FlagByte+随后的字节长度，仍是一个字节（byte），因此其最大值只能是0xff。即

> 极限情况（ FlagByte+随后字节长度 ） = 0xFF

另一方面，理论上我们编码支持的字节序列的最大长度应该为uint64类型的最大值，这最多需要8个字节，即

> 极限情况（ 随后字节长度所占字节 ） = 8

所以在极限情况下，即我想编码一个最大长度的list类型的字节序列时，“FlagByte+随后字节长度”也应达到最大值0xff。此时的标识值应该为0xff - 8，即0xf8。

> 1. 极限情况（ FlagByte+随后字节长度 ） = 0xFF
> 2. 极限情况（ 随后字节长度所占字节 ） = 8 以上两条 => 极限情况下 FlagByte+8 = 0xFF => 想要满足极限情况，FlagByte最大为0xF8，也应该恰好是这个值。

上面讨论的是编码字节的长度超过Threshold时的情况。在编码字节长度小于Threshold的情况下，0xF8应该是此时的极限值。

> FlagByte + Threshold = 0xF8

所以**ListFlagByte的值应该是"0xF8 - Threshold"**

现在我们再来考虑一下string的情况。其实这两种情况是类似的，区别就是极限情况下，即编码一个最大长度的string字节序列时，"FlagByte+随后字节长度"的最大值不是0xff，而是ListFlagByte，即"Threshold - 0xF8"。

> 极限情况（ FlagByte + 随后字节长度 ） = ListFlagByte

string的最大长度也是uint64类型的最大值，也需要8个字节。所以：

> 极限情况（ 随后字节长度所占字节 ） = 8

因此在极限情况下，标识值应该是ListFlagByte - 8:

> 1. 极限情况（ FlagByte + 随后字节长度 ） = ListFlagByte
> 2. 极限情况（ 随后字节长度所占字节 ） = 8
> 以上两条 => 极限情况下 FlagByte + 8 = ListFlagByte
> => 想要取得极限情况，FlagByte最大为ListFlagByte - 8

在string的长度小于Threshold的情况下，ListFlagByte - 8应该是此时的极限值：

> FlagByte + Threshold = ListFlagByte - 8

所以**StringFlagByte的值应该是：ListFlagByte - 8 - Threshold**，将**ListFlagByte**代入，得：**0xF0 - 2*Threshold**。

综上，现在我们StringFlagByte和ListFlagByte都知道了：

> StringFlagByte = 0xF0 - 2*Threshold
> ListFlagByte = 0xF8 - Threshold

但这里还是掺杂了一个变量Threshold。这个值要怎么确定呢？其实这个值的确定跟Single（可以直接编码的值的范围）有关系。StringFlagByte的意义在于，对于小于它的单个字节，可以直接编码，所以其实Single也就是StringFlagByte。所以如果你想让可以直接编码的单个字节的范围大一些，根据上面得到的"StringFlagByte = 0xF0 - 2*Threshold"这个式子，就可以让Threshold的值小一些；相反，如果你想让直接编码的单个字节的范围小些、而让简单编码的长度大一些，就可以让Threshold的值大一些。可见，Threshold这个值是可以根据情况任意选择和调整的。在rlp的实现里，Threshold的值选择为56，所以：

> Threshold = 56 StringFlagByte = 0x80 ListFlagByte = 0xC0

这与我们前面介绍的是一致的。

rlp源代码

rlp的代码位于go-ethereum项目下的rlp目录下。知道了rlp的编码规则以后，再来看源代码就会轻松很多。

上面我们已经提到，rlp只针对string和list进行编码。但被编码和解码的类型有各种各样的类型，包括自定义的结构体。所以源码里很大一部分工作都是在将这些各种类型转换成rlp的string或list。下面我们从多个角度逐一解析。

使用方法

首先我们来看一下怎么用这个rlp模块。这个模块的功能就是对数据结构进行序列化和反序列化。因此使用方法也从这两个功能入手分别介绍。

在使用之前，有几个问题是你需要注意的：

1. rlp不支持有符号整数的序列化和反序列化。另外虽然支持big.Int类型，但如果其值不能为负。这些问题在代码中的体现会在"实现框架"中进行说明。
2. 如果你要序列化或反序列化一个struct，那么它的字段必须是导出的（首字符大写）。这是因为rlp使用reflect包查看struct的字段。

序列化方法

rlp导出了几个函数和一个接口用于数据结构的序列化：

- EncodeToBytes(val interface{}) ([]byte, error)
  这个函最常用到。从名字上我们可以看到，此函数将一个结构体序列化后，返回其bytes数据。如：

```
val := struct {
        A string
        B uint32
}{
        "hello",
        0x32,
}

enc, _ := rlp.EncodeToBytes(val)
fmt.Printf("% x\n", enc)
////////////////////////////////////////
// output:
// c7 85 68 65 6c 6c 6f 32
```

- Encode(w io.Writer, val interface{})
  这个函数把数据结序列化以后，将结果写到w参数中。

- EncodeToReader(val interface{}) (size int, r io.Reader, err error)
  这个函数将数据结构序列化以后，返回一个io.Reader。调用者可以随时调用其Read文件将序列化结果取出。

- Encoder interface
  对于某些rlp不支持的类型（比如int），或者你想对一些数据结构自己进行特殊的编码时，都可以实现这个接口。在使用实现了这个接口的数据结构调用上面的函数时，rlp模块会发现并优先调用你实现的接口函数，而不再使用rlp自己的编码。

反序列化方法

反序列化的功能也是通过几个简单的导出函数和接口来实现的：

- DecodeBytes(b []byte, val interface{})
  这个函数是反序列化时最常用的函数。从名字可以看出，它将一串字节重新反序列化为某个数据结构变量。示例代码如下：

```
var val struct {
        A string
        B uint32
}
rlp.DecodeBytes([]byte{0xc7, 0x85, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x32}, &val)
fmt.Printf("%s\n", val.A)
fmt.Printf("0x%x\n", val.B)
////////////////////////////////////////
// output:
// hello
// 0x32
```

- Decode(r io.Reader, val interface{})
  与DecodeBytes类似，只不过这个函将从参数r中读取数据。

- Decoder interface
  与Encoder接口类似，都是为了实现自定义的反序列化操作。当你为某个类型实现了Encoder接口时，一般情况下你也要为其实现对应的Decoder接口。

结构体的tag

rlp支持3个tag，可以对结构体的字段进行不同的控制。

- rlp:"nil"
  这个tag用于指针类型的字段。如果存在这个tag，则在反序列化时如果不存在指针指向的数据，则设置这个字段为nil；如果不设置这个tag，则必须存在指针指向的数据，否则反序列化失败。看个例子会更加清楚。
  以下是使用nil tag结果：

```
type NilStruct struct {
  C uint
}
type AStruct struct {
        A string
        B *NilStruct `rlp:"nil"`
}

var val AStruct
//enc为 AStruct{"hello", nil}序列化后的数据
enc := []byte{0xc7, 0x85, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0xC0}
err := rlp.DecodeBytes(enc, &val)
if err != nil {
  fmt.Println(err)
  return
}
fmt.Printf("%s\n", val.A)
fmt.Printf("0x%x\n", val.B)
////////////////////////////////////////
// output:
// hello
// 0x0
```

可以看到成功反序列化，val.B被设置为nil。

下面的例子和刚才的一样，除了AStruct.B没有使用nil tag：

```
type NilStruct struct {
  C uint
}
type AStruct struct {
        A string
        B *NilStruct
}

var val AStruct
//enc为 AStruct{"hello", nil}序列化后的数据
enc := []byte{0xc7, 0x85, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0xC0}
err := rlp.DecodeBytes(enc, &val)
if err != nil {
  fmt.Println(err)
  return
}
fmt.Printf("%s\n", val.A)
fmt.Printf("0x%x\n", val.B)
////////////////////////////////////////
// output:
// rlp: too few elements for main.NilStruct, decoding into (struct { A string; B *main.NilStruct
}).B
```

可见在不用nil tag的时候，如果指针的针为空，则反序列化失败。

- rlp:"tail"
  这个tag必须在结构体的最后一个字段上使用，且这个字段的类型必须是slice。我没有想到这个字段的应用场景，因此它的功能也不好描述。我们直接看一下源代码里的例子吧：

```
type structWithTail struct {
        A, B uint
        C    []uint `rlp:"tail"`
}

func ExampleDecode_structTagTail() {
        // In this example, the "tail" struct tag is used to decode lists of
        // differing length into a struct.
        var val structWithTail

        err := Decode(bytes.NewReader([]byte{0xC4, 0x01, 0x02, 0x03, 0x04}), &val)
        fmt.Printf("with 4 elements: err=%v val=%v\n", err, val)

        err = Decode(bytes.NewReader([]byte{0xC6, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06}), &val)
        fmt.Printf("with 6 elements: err=%v val=%v\n", err, val)

        // Note that at least two list elements must be present to
        // fill fields A and B:
        err = Decode(bytes.NewReader([]byte{0xC1, 0x01}), &val)
        fmt.Printf("with 1 element: err=%q\n", err)

        // Output:
        // with 4 elements: err=<nil> val={1 2 [3 4]}
        // with 6 elements: err=<nil> val={1 2 [3 4 5 6]}
        // with 1 element: err="rlp: too few elements for rlp.structWithTail"
}
```

从例子中可以看出，无论是4个字节（0xC4开头）还是6个字节（0xC6开头）的数据，将结构体中其它字段（A和B）反序列化以后，剩下的数据都反序列化到了C这一最后的字段中。（但我仍然不理解这有什么用处，因为虽然字节的长度有变化，但仍然需要第一个字节进行标识）

- rlp:"-"
  这个tag的作用和用法非常简单，如果结构体中某个字段存在这个tag，这个字段会在序列化和反序列化时忽略。

目录结构

rlp目录下的文件数量不多，去掉test、doc文件，只有4个文件。下面分别对其说明：

- encode.go
  如文件名所示，这个文件里主要包含了序列化时的代码和数据结构。对外导出了三个函数Encode、EncodeToBytes、EncodeToReader，和一个接口Encoder。

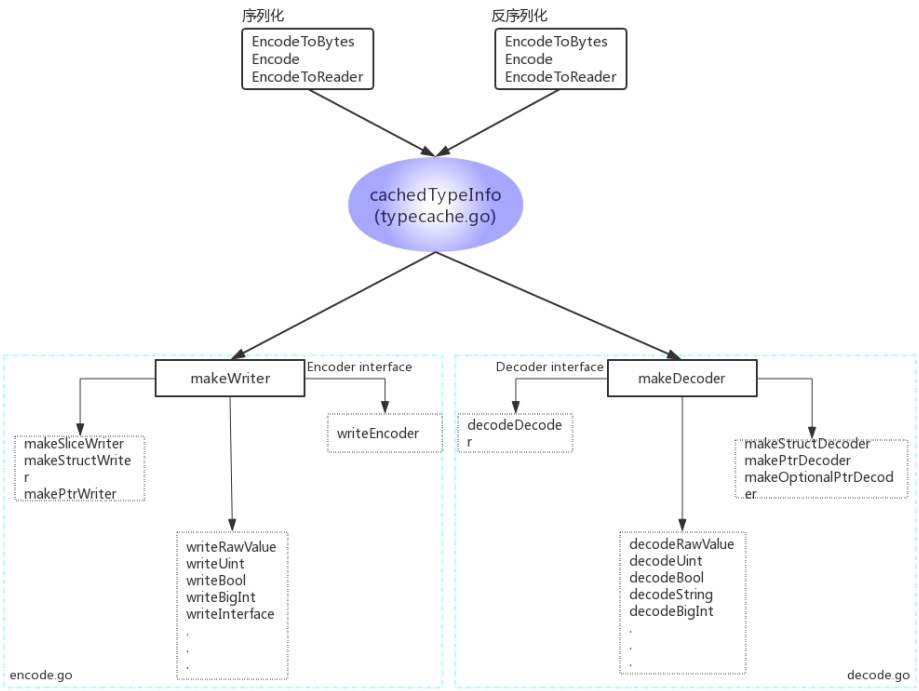
- decode.go
  如文件名所示，这个文件里主要包含了反序列化时的代码和数据结构。对导出了两个函数Decode、DecodeBytes，和一个接口Decoder。

- typecache.go
  这个文件中的代码主要实现了类型信息的生成和缓存功能。类型信息主要包括writer函数和decoder函数，以及结构体的tag信息。writer函数是用来对这个类型进行序列化的函数，decoder函数是对应的对这个类型进行反序列化的函数。

- raw.go
  这个文件中主要定义了RawValue类型，用来代表一段已经使用rlp编码好的数据。其本质是[]byte类型。另外还定义了几个辅助函数，用来帮助解析RawValue。这个文件中的代码作用不大且代码较少，因此后面的分析直接忽略了这个文件里的内容。

实现框架

从目录结构上我们可以看到，rlp的整体结构比较简单清晰。这里我们使用一张图来呈现代码的整个框架。



从框架图可以看出：

- cachedTypeInfo是枢纽，无论是序列化还是反序列化都需要从它这里拿到类型信息。
- makeWriter和makeDecoder是关键函数，这两个函数根据不同的类型，返回具体的序列化或反序列化函数。 cachedTypeInfo就是使用这两个函数获取writer和decoder的。
  另外如果我们查看makeWriter和makeDecoder的代码可以看到，这俩函数没有对有符号整数类型进行支持（switch/case中只有isUint的判断，没有isInt的判断）。
- 各种writeXXX和decodeXXX函数是序列化或反序列化某个类型的具体实现。个别较复杂的类型的具体实现是在makeXXXDecoder/makeXXXWriter中以lambda函数的方式实现的。 这里要特别注意一下writeBigInt函数，显然代码作者特意加了对big.Int这一类型的支持，但如果其具体的值为负时，仍然是不支持的。

细节分析

虽然rlp整体代码是比较清晰，但仍有几个点我认为不是那么一目了解，因此在这一小节中特意详细说明一下。

encbuf.lheads

encbuf是在序列化时使用的一结构体，主要用来逐一存储序列化以后的数据。其结构如下：

```
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 [9]byte     // 9-byte auxiliary buffer for uint encoding
}
```

这里重点要说的是encbuf.lheads这个字段。lheads字段是一个slice，每个元素是一个listhead指针：

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

当一段数据被序列化以后，就会被追加到encbuf.str中。而lheads这个字段的作用就是记录每一个list(还记得rlp中list的定义吧？)在encbuf.str中的起始位置和大小。也就是说，在encbuf.str中是没有0xC0或0xCF起始的标志字节和后面跟的数据长度的信息的，只有list的数据。在最后生成完整的序列化数据时（参见encbuf.toBytes），才跟据lheads中的信息，加入0xC0或0xCF标记字节。

为什么要这么实现呢？为什么不能直接把数据写到encbuf.str中，待到调用encbuf.toBytes时直接拷贝encbuf.str一下就可以了呢？我猜想是因为需要编码成list的数据结构都是复杂的结构类型，比如struct。在编码一个struct的开始时，是无法确定会往encbuf.str中新写入多长的数据的，只有编码完这个struct时才知道。因此作者才搞了一个lheads记录这个信息，在编码一个struct结束后把这个struct编码的起始位置和长度放入lheads中保存。简化的源代码如下：

```
func makeStructWriter(typ reflect.Type) (writer, error) {
        fields, err := structFields(typ)
        if err != nil {
                return nil, err
        }

        //编码struct的函数
        writer := func(val reflect.Value, w *encbuf) error {
                //encbuf.list函数会生成一个新的listhead，将其加入到lheads中并返回
                lh := w.list()

                for _, f := range fields {
                        f.info.writer(val.Field(f.index)
                }

                //结构体编码结束，更新lh里的信息。
                w.listEnd(lh)
                return nil
        }

        return writer, nil
}
```

在生成最终的序列化数据时，比如encbuf.toBytes函数时进行“拼接”：

```
func (w *encbuf) toBytes() []byte {
        out := make([]byte, w.size())
        strpos := 0
        pos := 0

        for _, head := range w.lheads {
                //在第一次执行for循环时，如果在第一个list之前有数据，先把它们拷贝到out中
                //在第一次之后的执行时，这里实际上拷贝的是上一次for循环生成的list head后面的数据。
                n := copy(out[pos:], w.str[strpos:head.offset])
                pos += n
                strpos += n

                //将head编码进out。
                enc := head.encode(out[pos:])
                pos += len(enc)
        }
```

```
        //将最后一个head后面的数据拷贝到out
        copy(out[pos:], w.str[strpos:])
        return out
}
```

这样实现虽然稍麻烦，但好处是在生成list没有数据的拷贝，只有到最后生成完整的序列化数据时拷贝并“拼接”起来最终的数据即可。

Stream.stack

Stream是在反序列化时使用的结构体。其中有一个字段stack，类型为[]listpos，listpos的定义非常简单：

```
type listpos struct{ pos, size uint64 }
```

如名字所示，这是一个栈结构，其元素的进出都是从尾部进行。这个字段的作用主要是在反序列化时作合规性检查。在反序列化一个list时（比如struct），调用Stream.List()记录当前这个list的起始位置和大小，并将信息push到Stream.stack中； 在反序列化这个list结束时，调用Stream.ListEnd()将栈顶的元素弹出（就是我们在开始时保存的信息）,并检查当前位置是否超出了我们记录的位置，如果超出，说明数据非法，可以立即返回错误。

之所以使用栈，是因为list是会有嵌套的。在处理一个list过程中，又遇到一个list需要处理，使用栈这种结构是非常合理的。

总结

rlp是以太坊中主要的序列化反序列化模块，其编码后的数据非常紧凑，同样数据编码后比json等格式的编码长度小很多。rlp中有两种结构：string和list，分别使用0x80和0xC0作为起始标志值，并跟据随后数据长度的大小（是否超过56字节）使用不同的方法将长度编码。

rlp模块的源文件很清晰，主要功能分在encode.go和decode.go中。在encode.go中包含了序列化相关的功能，对外导出了EncodeToBytes等函数和接口； decode.go中包含了反序列化相关的功能，对外导出了DecodeBytes等函数和接口。

本篇文章主要分析了rlp的编码方式，以及源码的大的框架。源代码中的函数命名还是非常清晰的，比如decodeXXX和writeXXX这些函数，一看就知道是编码或解码哪种数据类型的，因此在遇到问题时想详细查看某块功能时，可以快速找到相关实现。

以上分析有不对的地方，还望大家不吝指正。

Similar Posts

- 初识联盟链1: Fabric 是什么
- PBFT代码篇：fabric 中的 PBFT 实现
- 实用拜占庭容错算法（PBFT）
- 拜占庭将军问题
- 数据结构与算法：B树
- 以太坊源码解析：evm

上一篇 golang常量

下一篇 golang:反射的规则

Comments

Comments

Login

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name

Displayed next to your comments.

Email

Not displayed publicly.

Website (optional)

If you have a website, link to it here.

Subscribe to None

Submit Comment

Contact me at:

Site powered by Jekyll & Github Pages. Theme designed by HyG.