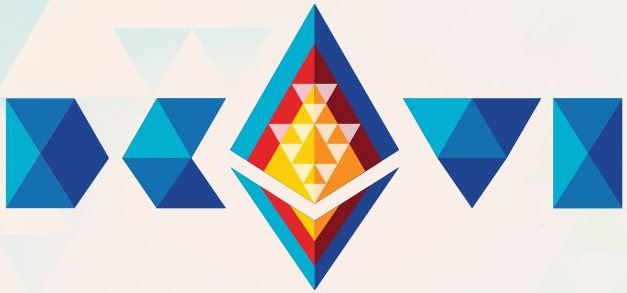


headlong: A Contract ABI and RLP Library for the JVM

An adventure in modular design and high performance

Evan Saulpaugh

github.com/esaulpaugh

github.com/esaulpaugh/headlong

What is headlong?

- An implementation of the ABIv2 and RLP specifications, and a few other encoding tools
- Built the Java way (but hopefully not like FizzBuzzEnterpriseEdition)

headlong is used by

- dune.com (Dune Analytics)
- unstoppabledomains/resolution-java
- Hedera Hashgraph
- and others

Section 1

The Idea

What do developers want in an encoding library?

1. Correctness. Working in 99.9% of cases is not an option.
    - Must be a full and faithful implementation of the RLP and ABI specifications
    - The headlong project makes extensive use of randomly-generated test cases, as well as known test vectors
2. Performance
    - High throughput
    - Low latency
3. A sensible API
    - I chose object-oriented design for intuitiveness and discoverability of functionality

Benchmark	Mode	Cnt	Score	Error	Units
MeasureFunction.decode_call	avgt	3	409.035	± 20.246	ns/op
MeasureFunction.decode_index_fast	avgt	3	33.730	± 4.502	ns/op
MeasureFunction.decode_index_slow	avgt	3	291.304	± 30.409	ns/op
MeasureFunction.encode_call	avgt	3	230.604	± 10.429	ns/op
MeasureFunction.init_function	avgt	3	885.275	± 58.106	ns/op
MeasureFunction.parse_tuple_type	avgt	3	350.549	± 6.465	ns/op

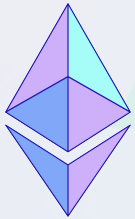
OpenJDK 8u345 for Windows, using JMH

Keep it focused

Do a small number of things well.

In this case:

- No I/O
- No cryptography (aside from some Keccak hashing)
- Just encoding/decoding
- Keep it low-level
- Very few dependencies (just gson)
- Don’t hyperfocus on one type of user or application



Benefits from modularity

- Enable rapid and continuous improvement through refactoring, redesign, and versioning
  - Without breaking everything!
  - Hopefully loose coupling.
    - **Encapsulation helps**
- Make something small and isolated enough for one person or a small team to manage
  - Express a consistent vision and design philosophy throughout the project
    - **Reduce the cognitive burden on users (and contributors)**
  - Avoid design-by-committee
- A sufficiently-focused project probably won't become out-of-date after every hard fork

Maximize compatibility

Java bytecode can be used by many other languages

In this case, dropping Java 8 compatibility didn't net any noticeable performance benefit, so headlong remains Java 8+.

Having only one runtime dependency makes headlong easy to integrate into a project.

Section 2

The Design

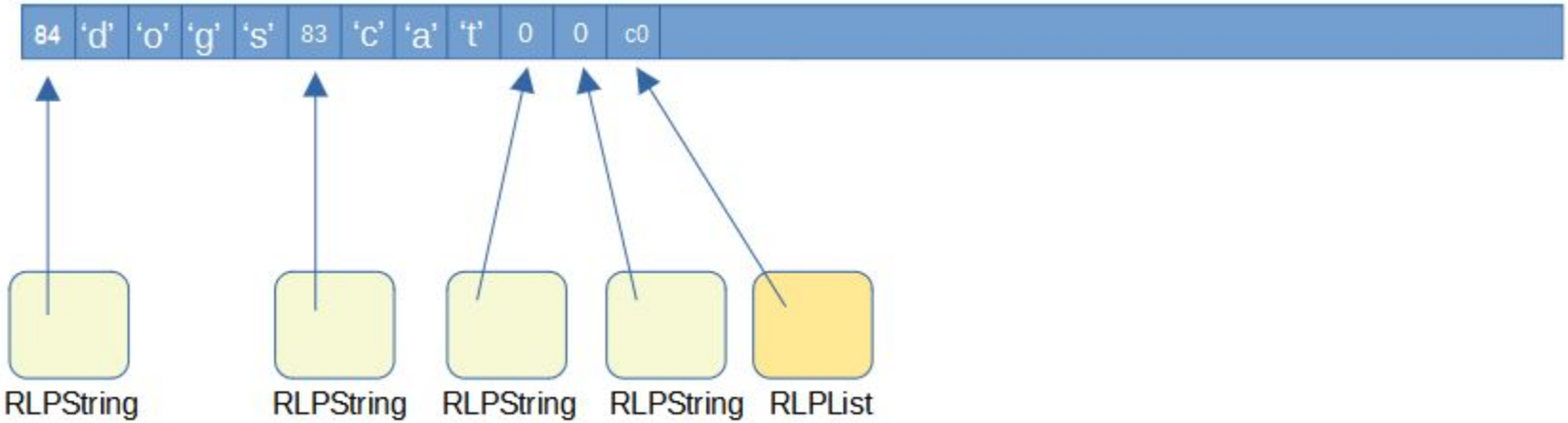
How it looks: RLP

```
Iterator<RLPItem> iter = RLP_STRICT.sequenceIterator(buffer);

iter.next(); // skip item
iter.next(); // skip item
String str    = iter.next().asString(UTF_8);
int number    = iter.next().asInt();
boolean bool = iter.next().asBoolean();
```

```
byte[] rlp = RLPEncoder.sequence(
        Integers.toBytes( val: 255),
        Strings.decode(name, UTF_8),
        FloatingPoint.toBytes(gpa),
        publicKeyBytes,
        balanceInCents.toByteArray()
);
```

How it works: RLP



How it looks: ABI

```
Function f = new Function( signature: "baz(uint32,bool)"); // canonicalizes and parses any signature

ByteBuffer call0 = f.encodeCallWithArgs(69L, true);
ByteBuffer call1 = f.encodeCallWithArgs(11L, true);
ByteBuffer call2 = f.encodeCallWithArgs(200L, false);
```

```
Function f2 = new Function( signature: "nextValue()",  outputs: "(int)");

BigInteger result = f2.decodeSingletonReturn(bytes);
```

You can also create a Function with Function.fromJson

How it works: ABI

- ABIObjects (Function, Event, ContractError) and ABITypes (ArrayType, TupleType, IntType, etc.) should be optimized for reuse
  - Precalculate as much as possible
- Use java.nio.ByteBuffer
  - Encode directly to raw bytes, not hexadecimal Strings
  - Much better performance
- Be stateless as much as possible
  - Allows sharing of objects across threads
  - Allows sharing of ABIType objects between Functions
    - One reusable canonical type object for each commonly-used type
- Implement some denial-of-service protection for when decoding untrusted/malformed data
  - We know that small calldatas cannot contain huge arrays

```
private static final ArrayType<ByteType, byte[]> BYTES_32 = TypeFactory.create("bytes32");
```

Some implementation details

The byte length of the head of an element is known ahead of time. For dynamically-size elements, it is the length of the offset which is always 32 bytes. For statically-sized elements, the head length can be recursively calculated from the type information alone.

Pre-calculating this head length allows headlong to skip tuple indices efficiently.

```
static int staticArrayHeadLength(ArrayType<?, ?> at) {
    switch (at.elementType.typeCode()) {
    case TYPE_CODE_BYTE: return UNIT_LENGTH_BYTES; // all static byte arrays round up to exactly 32 bytes and not more
    case TYPE_CODE_ARRAY: return at.length * staticArrayHeadLength((ArrayType<?, ?>) at.elementType);
    case TYPE_CODE_TUPLE: return at.length * TupleType.staticTupleHeadLength((TupleType) at.elementType);
    default: return at.length * UNIT_LENGTH_BYTES;
    }
}
```

Section 3

Future Work

Selective decode for nested tuples/structs?

The current API only allows skipping of top-level elements. It also doesn't enable partial decoding of arrays.

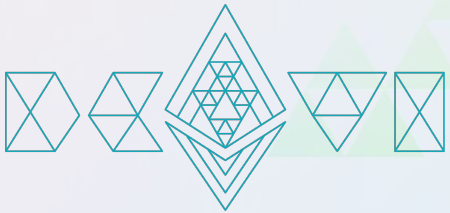
```
private Tuple decodeIndices(ByteBuffer bb, int... indices) {
    final Object[] results = new Object[elementTypes.length];
    final int start = bb.position();
    final byte[] unitBuffer = newUnitBuffer();
    for (int position = start, current = -1, i = 0; i < indices.length; i++) {
        final int index = indices[i];
        final ABIType<?> resultType = elementTypes[index]; // implicit bounds check up front
        if (index <= current) {
            throw new IllegalArgumentException("index out of order: " + index);
        }
        while (++current < index) {
            position += elementTypes[current].headLength();
        }
        bb.position(position);
        results[index] = decodeObject(resultType, bb, start, unitBuffer, index);
        position += resultType.headLength();
    }
    return new Tuple(results);
}
```

ABIv3?

Given a schema of the expected data types (e.g. ABI JSON), it is possible to encode and decode call data using RLP. This would cost fewer bytes compared to ABIv2, even if zero-bytes are considered free.

Offsets and padding are not necessary for unambiguous encoding. A tuple (string,string) with value (“abcd”, “efg”) could be encoded as 0x8461626364836566 67, nine bytes.

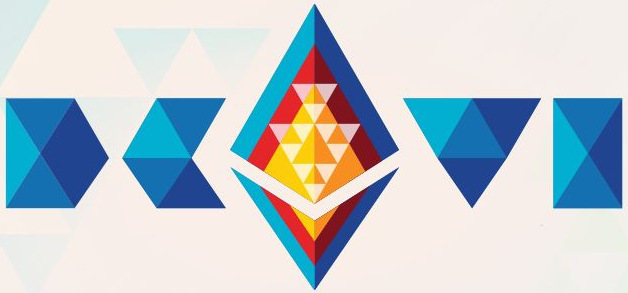
You can try this RLP representation by using the SuperSerial class or with the command-line interface project headlong-cli.



Faster, simpler software legos

Special thanks to

- Victor Delépine for contributing the event-decoding functionality
  - github.com/devictr



Thank you!

Evan Saulpaugh
github.com/esaulpaugh
evan.saulpaugh@gmail.com