Data Encoding for Class Type IDs

Type ID Encoding version 1.0

Unlike for exception type IDs, class type IDs are not simple strings. Instead, a class type ID is marshaled as a boolean followed by either a string or a size, to conserve bandwidth. To illustrate this, consider the following class hierarchy:

Slice

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
};
```

The type IDs for the class slices are `::Derived` and `::Base`. Suppose the sender marshals three instances of `::Derived` as part of a single request. (For example, two instances could be out-parameters and one instance could be the return value.)

The first instance that is sent on the wire contains the type IDs `::Derived` and `::Base` preceding their respective slices. Because marshaling proceeds in derived-to-base order, the first type ID that is sent is `::Derived`. Every time the sender sends a type ID that it has not sent previously in the same request, it sends the boolean value `false`, followed by the type ID. Internally, the sender also assigns a unique positive number to each type ID. These numbers start at `1` and increment by one for each type ID that has not been marshaled previously. This means that the first type ID is encoded as the boolean value `false`, followed by `::Derived`, and the second type ID is encoded as the boolean value `false`, followed by `::Base`.

When the sender marshals the remaining two instances, it consults a lookup table of previously-marshaled type IDs. Because both type IDs were sent previously in the same request (or reply), the sender encodes all further occurrences of `::Derived` as the value `true` followed by the number `1` encoded as a size, and it encodes all further occurrences of `::Base` as the value `true` followed by the number 2 encoded as a size.

When the receiver reads a type ID, it first reads its boolean marker:

- If the boolean is `false`, the receiver reads a string and enters that string into a lookup table that maps integers to strings. The first new class type ID received in a request is numbered `1`, the second new class type ID is numbered 2, and so on.
- If the boolean value is `true`, the receiver reads a number encoded as a size and uses it to retrieve the corresponding class type ID from the lookup table.

Note that this numbering scheme is re-established for each new encapsulation. (As we will see in our discussion of protocol messages, parameters, return values, and exceptions are always marshaled inside an enclosing encapsulation.) For subsequent or nested encapsulation, the numbering scheme restarts, with the first new type ID being assigned the value `1`. In other words, each encapsulation uses its own independent numbering scheme for class type IDs to satisfy the constraint that encapsulations must not depend on their surrounding context.

Encoding class type IDs in this way provides significant savings in bandwidth: whenever an ID is marshaled a second and subsequent time, it is marshaled as a two-byte value (assuming no more than 254 distinct type IDs per request) instead of as a string. Because type IDs can be long, especially if you are using nested modules, the savings are considerable.

Type ID Encoding version 1.1

Each slice of a class instance has a leading byte containing flags that describe various aspects of the slice, including whether the slice includes a type ID and how that type ID is encoded. There are four possibilities:

1. No type ID included
2. Type ID is encoded as a string
3. Type ID is encoded as an index
4. Type ID is encoded as a compact ID

The initial slice of an instance, representing the most-derived type, always contains some form of type ID so that the receiver knows what Slice type is present. Depending on the format used by the sender, subsequent slices may or may not include a type ID: the compact format omits type IDs in subsequent slices, whereas the sliced format includes a type ID in every slice. A receiver need only examine the slice flags to discover how to decode the type ID.

String Type IDs

The encoding for string type IDs uses a "compression" scheme similar to that of version 1.0: within an encapsulation, a given type ID is never encoded as a string more than once. The first time a sender encounters a type ID, the sender assigns an integer index to the ID and encodes the ID as a string. For all subsequent occurrences of the same type ID within the encapsulation, the sender encodes the index associated with that type ID as a size. Index values start at `1` and increase sequentially with each new type ID. The sender is responsible for setting the relevant bits in the flags of each slice to specify how the type ID is encoded.

The slice flags in version 1.1 of the encoding serve the same purpose as the boolean value that precedes each type ID in version 1.0, without consuming an entire byte.

Compact Type IDs

As a way of further reducing the overhead associated with class instances, version 1.1 adds the ability to substitute numeric values for strings when encoding type IDs. Consider the following Slice definitions:

Slice

```
class Base(3) {
    // ...
};

class Derived(4) extends Base {
    // ...
};

class MoreDerived extends Derived {
    // ...
};
```

The value in parentheses after the class name represents the *compact* type ID for the class. The sender's format determines whether each slice includes a type ID. If a given slice includes a type ID, the encoding always uses a compact type ID (if defined) in preference to its string equivalent. Suppose a sender is encoding an instance of `MoreDerived` in the sliced format. The initial slice uses the string type ID for `MoreDerived`, the next slice uses the compact type ID for `Derived`, and the last slice uses the compact type ID for `Base`.

A compact type ID is encoded as a size, with the relevant bits set in the slice flags.

The developer is responsible for ensuring compact type IDs are sufficiently unique. Using values less than 255 produces the most efficient encoding.

See Also

- Type IDs
- Basic Data Encoding
- Protocol Messages