Package gob

```
import "encoding/gob"
```

Overview ▾

Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver). A typical use is transporting arguments and results of remote procedure calls (RPCs) such as those provided by package "net/rpc".

The implementation compiles a custom codec for each data type in the stream and is most efficient when a single Encoder is used to transmit a stream of values, amortizing the cost of compilation.

Basics

A stream of gobs is self-describing. Each data item in the stream is preceded by a specification of its type, expressed in terms of a small set of predefined types. Pointers are not transmitted, but the things they point to are transmitted; that is, the values are flattened. Nil pointers are not permitted, as they have no value. Recursive types work fine, but recursive values (data with cycles) are problematic. This may change.

To use gobs, create an Encoder and present it with a series of data items as values or addresses that can be dereferenced to values. The Encoder makes sure all type information is sent before it is needed. At the receive side, a Decoder retrieves values from the encoded stream and unpacks them into local variables.

Types and Values

The source and destination values/types need not correspond exactly. For structs, fields (identified by name) that are in the source but absent from the receiving variable will be ignored. Fields that are in the receiving variable but missing from the transmitted type or value will be ignored in the destination. If a field with the same name is present in both, their types must be compatible. Both the receiver and transmitter will do all necessary indirection and dereferencing to convert between gobs and actual Go values. For instance, a gob type that is schematically,

```
struct { A, B int }
```

can be sent from or received into any of these Go types:

```
struct { A, B int }     // the same
*struct { A, B int }    // extra indirection of the struct
struct { *A, **B int }  // extra indirection of the fields
struct { A, B int64 }   // different concrete value type; see below
```

It may also be received into any of these:

```
struct { A, B int }     // the same
struct { B, A int }     // ordering doesn't matter; matching is by name
struct { A, B, C int }  // extra field (C) ignored
struct { B int }        // missing field (A) ignored; data will be dropped
struct { B, C int }     // missing field (A) ignored; extra field (C) ignored.
```

Attempting to receive into these types will draw a decode error:

```
struct { A int; B uint }        // change of signedness for B
struct { A int; B float }       // change of type for B
struct { }                      // no field names in common
struct { C, D int }             // no field names in common
```

Integers are transmitted two ways: arbitrary precision signed integers or arbitrary precision unsigned integers. There is no int8, int16 etc. discrimination in the gob format; there are only signed and unsigned integers. As described below, the transmitter sends the value in a variable-length encoding; the receiver accepts the value and stores it in the destination variable. Floating-point numbers are always sent using IEEE-754 64-bit precision (see below).

Signed integers may be received into any signed integer variable: int, int16, etc.; unsigned integers may be received into any unsigned integer variable; and floating point values may be received into any floating point variable. However, the destination variable must be able to represent the value or the decode operation will fail.

Structs, arrays and slices are also supported. Structs encode and decode only exported fields. Strings and arrays of bytes are supported with a special, efficient representation (see below). When a slice is decoded, if the existing slice has capacity the slice will be extended in place; if not, a new array is allocated. Regardless, the length of the resulting slice reports the number of elements decoded.

In general, if allocation is required, the decoder will allocate memory. If not, it will update the destination variables with values read from the stream. It does not initialize them first, so if the destination is a compound value such as a map, struct, or slice, the decoded values will be merged elementwise into the existing variables.

Functions and channels will not be sent in a gob. Attempting to encode such a value at the top level will fail. A struct field of chan or func type is treated exactly like an unexported field and is ignored.

Gob can encode a value of any type implementing the GobEncoder or encoding.BinaryMarshaler interfaces by calling the corresponding method, in that order of preference.

Gob can decode a value of any type implementing the GobDecoder or encoding.BinaryUnmarshaler interfaces by calling the corresponding method, again in that order of preference.

Encoding Details

This section documents the encoding, details that are not important for most users. Details are presented bottom-up.

An unsigned integer is sent one of two ways. If it is less than 128, it is sent as a byte with that value. Otherwise it is sent as a minimal-length big-endian (high byte first) byte stream holding the value, preceded by one byte holding the byte count, negated. Thus 0 is transmitted as (00), 7 is transmitted as (07) and 256 is transmitted as (FE 01 00).

A boolean is encoded within an unsigned integer: 0 for false, 1 for true.

A signed integer, i, is encoded within an unsigned integer, u. Within u, bits 1 upward contain the value; bit 0 says whether they should be complemented upon receipt. The encode algorithm looks like this:

```
var u uint
if i < 0 {
	u = (^uint(i) << 1) | 1 // complement i, bit 0 is 1
} else {
	u = (uint(i) << 1) // do not complement i, bit 0 is 0
}
encodeUnsigned(u)
```

The low bit is therefore analogous to a sign bit, but making it the complement bit instead guarantees that the largest negative integer is not a special case. For example, -129=^128=(^256>>1) encodes as (FE 01 01).

Floating-point numbers are always sent as a representation of a float64 value. That value is converted to a uint64 using math.Float64bits. The uint64 is then byte-reversed and sent as a regular unsigned integer. The byte-reversal means the exponent and high-precision part of the mantissa go first. Since the low bits are often zero, this can save encoding bytes. For instance, 17.0 is encoded in only three bytes (FE 31 40).

Strings and slices of bytes are sent as an unsigned count followed by that many uninterpreted bytes of the value.

All other slices and arrays are sent as an unsigned count followed by that many elements using the standard gob encoding for their type, recursively.

Maps are sent as an unsigned count followed by that many key, element pairs. Empty but non-nil maps are sent, so if the receiver has not allocated one already, one will always be allocated on receipt unless the transmitted map is nil and not at the top level.

In slices and arrays, as well as maps, all elements, even zero-valued elements, are transmitted, even if all the elements are zero.

Structs are sent as a sequence of (field number, field value) pairs. The field value is sent using the standard gob encoding for its type, recursively. If a field has the zero value for its type (except for arrays; see above), it is omitted from the transmission. The field number is defined by the type of the encoded struct: the first field of the encoded type is field 0, the second is field 1, etc. When encoding a value, the field numbers are delta encoded for efficiency and the fields are always sent in order of increasing field number; the deltas are therefore unsigned. The initialization for the delta encoding sets the field number to -1, so an unsigned integer field 0 with value 7 is transmitted as unsigned delta = 1, unsigned value = 7 or (01 07). Finally, after all the fields have been sent a terminating mark denotes the end of the struct. That mark is a delta=0 value, which has representation (00).

Interface types are not checked for compatibility; all interface types are treated, for transmission, as members of a single "interface" type, analogous to int or []byte - in effect they're all treated as interface{}. Interface values are transmitted as a string identifying the concrete type being sent (a name that must be pre-defined by calling Register), followed by a byte count of the length of the following data (so the value can be skipped if it cannot be stored), followed by the usual encoding of concrete (dynamic) value stored in the interface value. (A nil interface value is identified by the empty string and transmits no value.) Upon receipt, the decoder verifies that the unpacked concrete item satisfies the interface of the receiving variable.

If a value is passed to Encode and the type is not a struct (or pointer to struct, etc.), for simplicity of processing it is represented as a struct of one field. The only visible effect of this is to encode a zero byte after the value, just as after the last field of an encoded struct, so that the decode algorithm knows when the top-level value is complete.

The representation of types is described below. When a type is defined on a given connection between an Encoder and Decoder, it is assigned a signed integer type id. When Encoder.Encode(v) is called, it makes sure there is an id assigned for the type of v and all its elements and then it sends the

pair (typeid, encoded-v) where typeid is the type id of the encoded type of v and encoded-v is the gob encoding of the value v.

To define a type, the encoder chooses an unused, positive type id and sends the pair (-type id, encoded-type) where encoded-type is the gob encoding of a wireType description, constructed from these types:

```
type wireType struct {
        ArrayT           *ArrayType
        SliceT           *SliceType
        StructT          *StructType
        MapT             *MapType
        GobEncoderT      *gobEncoderType
        BinaryMarshalerT *gobEncoderType
        TextMarshalerT   *gobEncoderType

}
type arrayType struct {
        CommonType
        Elem typeId
        Len  int
}
type CommonType struct {
        Name string // the name of the struct type
        Id  int    // the id of the type, repeated so it's inside the type
}
type sliceType struct {
        CommonType
        Elem typeId
}
type structType struct {
        CommonType
        Field []*fieldType // the fields of the struct.
}
type fieldType struct {
        Name string // the name of the field.
        Id   int    // the type id of the field, which must be already defined
}
type mapType struct {
        CommonType
        Key  typeId
        Elem typeId
}
type gobEncoderType struct {
        CommonType
}
```

If there are nested type ids, the types for all inner type ids must be defined before the top-level type id is used to describe an encoded-v.

For simplicity in setup, the connection is defined to understand these types a priori, as well as the basic gob types int, uint, etc. Their ids are:

basic gob types int, uint, etc. Their ids are:

```
bool        1
int         2
uint        3
float       4
[]byte      5
string      6
complex     7
interface   8
// gap for reserved ids.
WireType    16
ArrayType   17
CommonType  18
SliceType   19
StructType  20
FieldType   21
// 22 is slice of fieldType.
MapType     23
```

Finally, each message created by a call to Encode is preceded by an encoded unsigned integer count of the number of bytes remaining in the message. After the initial type name, interface values are wrapped the same way; in effect, the interface value acts like a recursive invocation of Encode.

In summary, a gob stream looks like

```
(byteCount (-type id, encoding of a wireType)* (type id, encoding of a value))*
```

where * signifies zero or more repetitions and the type id of a value must be predefined or be defined before the value in the stream.

Compatibility: Any future changes to the package will endeavor to maintain compatibility with streams encoded using previous versions. That is, any released version of this package should be able to decode data written with any previously released version, subject to issues such as security fixes. See the Go compatibility document for background: https://golang.org/doc/go1compat

See "Gobs of data" for a design discussion of the gob wire format: https://blog.golang.org/gobs-of-data

▹ Example (Basic)

▹ Example (EncodeDecode)

▹ Example (Interface)

Index ▾

Examples (Expand All)

Package files

dec_helpers.go decode.go decoder.go doc.go enc_helpers.go encode.go encoder.go error.go type.go

func Register

```
func Register(value interface{})
```

Register records a type, identified by a value for that type, under its internal type name. That name will identify the concrete type of a value sent or received as an interface variable. Only types that will be transferred as implementations of interface values need to be registered. Expecting to be used only during initialization, it panics if the mapping between types and names is not a bijection.

func RegisterName

```
func RegisterName(name string, value interface{})
```

RegisterName is like Register but uses the provided name rather than the type's default.

type CommonType

CommonType holds elements of all types. It is a historical artifact, kept for binary compatibility and exported only for the benefit of the package's encoding of type descriptors. It is not intended for direct use by clients.

```
type CommonType struct {
    Name string
    Id   typeId
}
```

type Decoder

A Decoder manages the receipt of type and data information read from the remote side of a connection. It is safe for concurrent use by multiple goroutines.

connection. It is safe for concurrent use by multiple goroutines.

The Decoder does only basic sanity checking on decoded input sizes, and its limits are not configurable. Take caution when decoding gob data from untrusted sources.

```
type Decoder struct {
    // contains filtered or unexported fields
}
```

func NewDecoder

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder returns a new decoder that reads from the io.Reader. If r does not also implement io.ByteReader, it will be wrapped in a bufio.Reader.

func (*Decoder) Decode

```
func (dec *Decoder) Decode(e interface{}) error
```

Decode reads the next value from the input stream and stores it in the data represented by the empty interface value. If e is nil, the value will be discarded. Otherwise, the value underlying e must be a pointer to the correct type for the next data item received. If the input is at EOF, Decode returns io.EOF and does not modify e.

func (*Decoder) DecodeValue

```
func (dec *Decoder) DecodeValue(v reflect.Value) error
```

DecodeValue reads the next value from the input stream. If v is the zero reflect.Value (v.Kind() == Invalid), DecodeValue discards the value. Otherwise, it stores the value into v. In that case, v must represent a non-nil pointer to data or be an assignable reflect.Value (v.CanSet()) If the input is at EOF, DecodeValue returns io.EOF and does not modify v.

type Encoder

An Encoder manages the transmission of type and data information to the other side of a connection. It is safe for concurrent use by multiple goroutines.

```
type Encoder struct {
    // contains filtered or unexported fields
}
```

func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that will transmit on the io.Writer.

func (*Encoder) Encode

```
func (enc *Encoder) Encode(e interface{}) error
```

Encode transmits the data item represented by the empty interface value, guaranteeing that all necessary type information has been transmitted first. Passing a nil pointer to Encoder will panic, as they cannot be transmitted by gob.

func (*Encoder) EncodeValue

```
func (enc *Encoder) EncodeValue(value reflect.Value) error
```

EncodeValue transmits the data item represented by the reflection value, guaranteeing that all necessary type information has been transmitted first. Passing a nil pointer to EncodeValue will panic, as they cannot be transmitted by gob.

type GobDecoder

GobDecoder is the interface describing data that provides its own routine for decoding transmitted values sent by a GobEncoder.

```
type GobDecoder interface {
    // GobDecode overwrites the receiver, which must be a pointer,
    // with the value represented by the byte slice, which was written
    // by GobEncode, usually for the same concrete type.
    GobDecode([]byte) error
}
```

type GobEncoder

GobEncoder is the interface describing data that provides its own representation for encoding values for transmission to a GobDecoder. A type that implements GobEncoder and GobDecoder has complete control over the representation of its data and may therefore contain things such as private fields, channels, and functions, which are not usually transmissible in gob streams.

Note: Since gobs can be stored permanently, it is good design to guarantee the encoding used by a GobEncoder is stable as the software evolves. For instance, it might make sense for GobEncode to include a version number in the encoding.

```
type GobEncoder interface {
    // GobEncode returns a byte slice representing the encoding of the
    // receiver for transmission to a GobDecoder, usually of the same
    // concrete type.
    GobEncode() ([]byte, error)
}
```