Package rpc go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package rpc provides access to the exported methods of an object across a network or other I/O connection. A server registers an object, making it visible as a service with the name of the type of the object. After registration, exported methods of the object will be accessible remotely. A server may register multiple objects (services) of different types but it is an error to register multiple objects of the same type.

Only methods that satisfy these criteria will be made available for remote access; other methods will be ignored:

```
- the method's type is exported.
- the method is exported.
- the method has two arguments, both exported (or builtin) types.
- the method's second argument is a pointer.
- the method has return type error.
```

In effect, the method must look schematically like

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

where T1 and T2 can be marshaled by encoding/gob. These requirements apply even if a different codec is used. (In the future, these requirements may soften for custom codecs.)

The method's first argument represents the arguments provided by the caller; the second argument represents the result parameters to be returned to the caller. The method's return value, if non-nil, is passed back as a string that the client sees as if created by errors.New. If an error is returned, the reply parameter will not be sent back to the client.

The server may handle requests on a single connection by calling ServeConn. More typically it will create a network listener and call Accept or, for an HTTP listener, HandleHTTP and http.Serve.

A client wishing to use the service establishes a connection and then invokes NewClient on the connection. The convenience function Dial (DialHTTP) performs both steps for a raw network connection (an HTTP connection). The resulting Client object has two methods, Call and Go, that specify the service and method to call, a pointer containing the arguments, and a pointer to receive the result parameters.

The Call method waits for the remote call to complete while the Go method launches the call asynchronously and signals completion using the Call structure's Done channel.

Unless an explicit codec is set up, package encoding/gob is used to transport the data.

Here is a simple example. A server wishes to export an object of type Arith:

```
package server

import "errors"

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

The server calls (for HTTP service):

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

At this point, clients can see a service "Arith" with methods "Arith.Multiply" and "Arith.Divide". To invoke one, a client first dials the server:

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

Then it can make a remote call:

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

or

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done // will be equal to divCall
// check errors, print, etc.
```

A server implementation will often provide a simple, type-safe wrapper for the client.

The net/rpc package is frozen and is not accepting new features.

Constants

```
const (
    // Defaults used by HandleHTTP
    DefaultRPCPath   = "/_goRPC_"
    DefaultDebugPath = "/debug/rpc"
)
```

Variables

```
var DefaultServer = NewServer()
```

DefaultServer is the default instance of *Server.

```
var ErrShutdown = errors.New("connection is shut down")
```

func Accept

```
func Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests to DefaultServer for each incoming connection. Accept blocks; the caller typically invokes it in a go statement.

func HandleHTTP

```
func HandleHTTP()
```

HandleHTTP registers an HTTP handler for RPC messages to DefaultServer on DefaultRPCPath and a debugging handler on DefaultDebugPath. It is still necessary to invoke http.Serve(), typically in a go statement.

func Register

```
func Register(rcvr interface{}) error
```

Register publishes the receiver's methods in the DefaultServer.

func RegisterName

```
func RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

func ServeCodec

```
func ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

func ServeConn

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the DefaultServer on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement. ServeConn uses the gob wire format (see package gob) on the connection. To use an alternate codec, use ServeCodec. See NewClient's comment for information about concurrent access.

func ServeRequest

```
func ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

type Call

```
type Call struct {
    ServiceMethod string      // The name of the service and method to call.
    Args          interface{} // The argument to the function (*struct).
    Reply         interface{} // The reply from the function (*struct).
    Error         error       // After completion, the error status.
    Done          chan *Call  // Receives *Call when Go is complete.
}
```

Call represents an active RPC.

type Client

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

Client represents an RPC Client. There may be multiple outstanding Calls associated with a single Client, and a Client may be used by multiple goroutines simultaneously.

func Dial

```
func Dial(network, address string) (*Client, error)
```

Dial connects to an RPC server at the specified network address.

func DialHTTP

```
func DialHTTP(network, address string) (*Client, error)
```

DialHTTP connects to an HTTP RPC server at the specified network address listening on the default HTTP RPC path.

func DialHTTPPath

```
func DialHTTPPath(network, address, path string) (*Client, error)
```

DialHTTPPath connects to an HTTP RPC server at the specified network address and path.

func NewClient

```
func NewClient(conn io.ReadWriteCloser) *Client
```

NewClient returns a new Client to handle requests to the set of services at the other end of the connection. It adds a buffer to the write side of the connection so the header and payload are sent as a unit.

The read and write halves of the connection are serialized independently, so no interlocking is required. However each half may be accessed concurrently so the implementation of conn should protect against concurrent reads or concurrent writes.

func NewClientWithCodec

```
func NewClientWithCodec(codec ClientCodec) *Client
```

NewClientWithCodec is like NewClient but uses the specified codec to encode requests and decode responses.

func (*Client) Call

```
func (client *Client) Call(serviceMethod string, args interface{}, reply interface{})
```

Call invokes the named function, waits for it to complete, and returns its error status.

func (*Client) Close

```
func (client *Client) Close() error
```

Close calls the underlying codec's Close method. If the connection is already shutting down, ErrShutdown is returned.

func (*Client) Go

```
func (client *Client) Go(serviceMethod string, args interface{}, reply interface{}, d
```

Go invokes the function asynchronously. It returns the Call structure representing the invocation. The done channel will signal when the call is complete by returning the same Call object. If done is nil, Go will allocate a new channel. If non-nil, done must be buffered or Go will deliberately crash.

type ClientCodec

```
type ClientCodec interface {
    WriteRequest(*Request, interface{}) error
    ReadResponseHeader(*Response) error
    ReadResponseBody(interface{}) error

    Close() error
}
```

A ClientCodec implements writing of RPC requests and reading of RPC responses for the client side of an RPC session. The client calls WriteRequest to write a request to the connection and calls ReadResponseHeader and ReadResponseBody in pairs to read responses. The client calls Close when finished with the connection. ReadResponseBody may be called with a nil argument to force the body

of the response to be read and then discarded. See NewClient's comment for information about concurrent access.

type Request

```
type Request struct {
    ServiceMethod string // format: "Service.Method"
    Seq           uint64 // sequence number chosen by client
    // contains filtered or unexported fields
}
```

Request is a header written before every RPC call. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

type Response

```
type Response struct {
    ServiceMethod string // echoes that of the Request
    Seq           uint64 // echoes that of the request
    Error         string // error, if any.
    // contains filtered or unexported fields
}
```

Response is a header written before every RPC return. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

type Server

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

Server represents an RPC Server.

func NewServer

```
func NewServer() *Server
```

NewServer returns a new Server.

func (*Server) Accept

```
func (server *Server) Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests for each incoming connection. Accept blocks until the listener returns a non-nil error. The caller typically invokes Accept in a go statement.

func (*Server) HandleHTTP

```
func (server *Server) HandleHTTP(rpcPath, debugPath string)
```

HandleHTTP registers an HTTP handler for RPC messages on rpcPath, and a debugging handler on debugPath. It is still necessary to invoke http.Serve(), typically in a go statement.

func (*Server) Register

```
func (server *Server) Register(rcvr interface{}) error
```

Register publishes in the server the set of methods of the receiver value that satisfy the following conditions:

```
- exported method of exported type
- two arguments, both of exported type
- the second argument is a pointer
- one return value, of type error
```

It returns an error if the receiver is not an exported type or has no suitable methods. It also logs the error using package log. The client accesses each method using a string of the form "Type.Method", where Type is the receiver's concrete type.

func (*Server) RegisterName

```
func (server *Server) RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

func (*Server) ServeCodec

```
func (server *Server) ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

func (*Server) ServeConn

```
func (server *Server) ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the server on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement. ServeConn uses the gob wire format (see package gob) on the connection. To use an alternate codec, use ServeCodec. See NewClient's comment for information about concurrent access.

func (*Server) ServeHTTP

```
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Request)
```

ServeHTTP implements an http.Handler that answers RPC requests.

func (*Server) ServeRequest

```
func (server *Server) ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

type ServerCodec

```
type ServerCodec interface {
    ReadRequestHeader(*Request) error
    ReadRequestBody(interface{}) error
    WriteResponse(*Response, interface{}) error

    // Close can be called multiple times and must be idempotent.
    Close() error
}
```

A ServerCodec implements reading of RPC requests and writing of RPC responses for the server side of an RPC session. The server calls ReadRequestHeader and ReadRequestBody in pairs to read requests from the connection, and it calls WriteResponse to write a response back. The server calls Close when finished with the connection. ReadRequestBody may be called with a nil argument to force the body of the request to be read and discarded. See NewClient's comment for information about concurrent access.

type ServerError

```
type ServerError string
```

ServerError represents an error that has been returned from the remote side of the RPC connection.

func (ServerError) Error

```
func (e ServerError) Error() string
```