textproto

package | standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 10 | Imported by: 10,281

Details Valid go.mod file Redistributable license Tagged version Stable version

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Documentation

Overview

Package textproto implements generic support for text-based request/response protocols in the style of HTTP, NNTP, and SMTP.

The package provides:

Error, which represents a numeric error response from a server.

Pipeline, to manage pipelined requests and responses in a client.

Reader, to read numeric response code lines, key: value headers, lines wrapped with leading spaces on continuation lines, and whole text blocks ending with a dot on a line by itself.

Writer, to write dot-encoded text blocks.

Conn, a convenient packaging of Reader, Writer, and Pipeline for use with a single network connection.

Index

Constants

This section is empty.

Variables

This section is empty.

Functions

func CanonicalMIMEHeaderKey

```
func CanonicalMIMEHeaderKey(s string) string
```

CanonicalMIMEHeaderKey returns the canonical format of the MIME header key s. The canonicalization converts the first letter and any letter following a hyphen to upper case; the rest are converted to lowercase. For example, the canonical key for "accept-encoding" is "Accept-Encoding". MIME header keys are assumed to be ASCII only. If s contains a space or invalid header field bytes, it is returned without modifications.

func TrimBytes

added in go1.1

```
func TrimBytes(b []byte) []byte
```

TrimBytes returns b without leading and trailing ASCII space.

func TrimString

added in go1.1

```
func TrimString(s string) string
```

TrimString returns s without leading and trailing ASCII space.

Types

type Conn

```
type Conn struct {
    Reader
    Writer
    Pipeline
    // contains filtered or unexported fields
}
```

A Conn represents a textual network protocol connection. It consists of a Reader and Writer to manage I/O and a Pipeline to sequence concurrent requests on the connection. These embedded types carry methods with them; see the documentation of those types for details.

func Dial

```
func Dial(network, addr string) (*Conn, error)
```

Dial connects to the given address on the given network using net.Dial and then returns a new Conn for the connection.

func NewConn

```
func NewConn(conn io.ReadWriteCloser) *Conn
```

NewConn returns a new Conn using conn for I/O.

func (*Conn) Close

```
func (c *Conn) Close() error
```

Close closes the connection.

func (*Conn) Cmd

```
func (c *Conn) Cmd(format string, args ...any) (id uint, err error)
```

Cmd is a convenience method that sends a command after waiting its turn in the pipeline. The command text is the result of formatting format with args and appending \r\n. Cmd returns the id of the command, for use with StartResponse and EndResponse.

For example, a client might run a HELP command that returns a dot-body by using:

```
id, err := c.Cmd("HELP")
if err != nil {
    return nil, err
}

c.StartResponse(id)
defer c.EndResponse(id)

if _, _, err = c.ReadCodeLine(110); err != nil {
    return nil, err
}
text, err := c.ReadDotBytes()
if err != nil {
    return nil, err
}
return c.ReadCodeLine(250)
```

type Error

```
type Error struct {
    Code int
    Msg  string
}
```

An Error represents a numeric error response from a server.

func (*Error) Error

```
func (e *Error) Error() string
```

type MIMEHeader

```
type MIMEHeader map[string][]string
```

A MIMEHeader represents a MIME-style header mapping keys to sets of values.

func (MIMEHeader) Add

```
func (h MIMEHeader) Add(key, value string)
```

Add adds the key, value pair to the header. It appends to any existing values associated with key.

func (MIMEHeader) Del

```
func (h MIMEHeader) Del(key string)
```

Del deletes the values associated with key.

func (MIMEHeader) Get

```
func (h MIMEHeader) Get(key string) string
```

Get gets the first value associated with the given key. It is case insensitive; CanonicalMIMEHeaderKey is used to canonicalize the provided key. If there are no values associated with the key, Get returns "". To use non-canonical keys, access the map directly.

func (MIMEHeader) Set

```
func (h MIMEHeader) Set(key, value string)
```

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key.

func (MIMEHeader) Values

added in go1.14

```
func (h MIMEHeader) Values(key string) []string
```

Values returns all values associated with the given key. It is case insensitive; CanonicalMIMEHeaderKey is used to canonicalize the provided key. To use non-canonical keys, access the map directly. The returned slice is not a copy.

type Pipeline

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

A Pipeline manages a pipelined in-order request/response sequence.

To use a Pipeline p to manage multiple clients on a connection, each client should run:

```
id := p.Next()  // take a number

p.StartRequest(id)  // wait for turn to send request
```

```
«send request»
p.EndRequest(id)    // notify Pipeline that request is sent

p.StartResponse(id) // wait for turn to read response
«read response»
p.EndResponse(id)   // notify Pipeline that response is read
```

A pipelined server can use the same calls to ensure that responses computed in parallel are written in the correct order.

func (*Pipeline) EndRequest

```
func (p *Pipeline) EndRequest(id uint)
```

EndRequest notifies p that the request with the given id has been sent (or, if this is a server, received).

func (*Pipeline) EndResponse

```
func (p *Pipeline) EndResponse(id uint)
```

EndResponse notifies p that the response with the given id has been received (or, if this is a server, sent).

func (*Pipeline) Next

```
func (p *Pipeline) Next() uint
```

Next returns the next id for a request/response pair.

func (*Pipeline) StartRequest

```
func (p *Pipeline) StartRequest(id uint)
```

StartRequest blocks until it is time to send (or, if this is a server, receive) the request with the given id.

func (*Pipeline) StartResponse

```
func (p *Pipeline) StartResponse(id uint)
```

StartResponse blocks until it is time to receive (or, if this is a server, send) the request with the given id.

type ProtocolError

```
type ProtocolError string
```

A ProtocolError describes a protocol violation such as an invalid response or a hung-up connection.

func (ProtocolError) Error

```
func (p ProtocolError) Error() string
```

type Reader

```
type Reader struct {
    R *bufio.Reader
    // contains filtered or unexported fields
}
```

A Reader implements convenience methods for reading requests or responses from a text protocol network connection.

func NewReader

```
func NewReader(r *bufio.Reader) *Reader
```

NewReader returns a new Reader reading from r.

To avoid denial of service attacks, the provided bufio.Reader should be reading from an io.LimitReader or similar Reader to bound the size of responses.

func (*Reader) DotReader

```
func (r *Reader) DotReader() io.Reader
```

DotReader returns a new Reader that satisfies Reads using the decoded text of a dot-encoded block read from r. The returned Reader is only valid until the next call to a method on r.

Dot encoding is a common framing used for data blocks in text protocols such as SMTP. The data consists of a sequence of lines, each of which ends in "\r\n". The sequence itself ends at a line containing just a dot: ".\r\n". Lines beginning with a dot are escaped with an additional dot to avoid looking like the end of the sequence.

The decoded form returned by the Reader's Read method rewrites the "\r\n" line endings into the simpler "\n", removes leading dot escapes if present, and stops with error io.EOF after consuming (and discarding) the end-of-sequence line.

func (*Reader) ReadCodeLine

```
func (r *Reader) ReadCodeLine(expectCode int) (code int, message string, err error)
```

ReadCodeLine reads a response code line of the form

```
code message
```

where code is a three-digit status code and the message extends to the rest of the line. An example of such a line is:

```
220 plan9.bell-labs.com ESMTP
```

If the prefix of the status does not match the digits in expectCode, ReadCodeLine returns with err set to &Error{code, message}. For example, if expectCode is 31, an error will be returned if the status is not in the range [310,319].

If the response is multi-line, ReadCodeLine returns an error.

An expectCode <= 0 disables the check of the status code.

func (*Reader) ReadContinuedLine

```
func (r *Reader) ReadContinuedLine() (string, error)
```

ReadContinuedLine reads a possibly continued line from r, eliding the final trailing ASCII white space. Lines after the first are considered continuations if they begin with a space or tab character. In the returned data, continuation lines are separated from the previous line only by a single space: the newline and leading white space are removed.

For example, consider this input:

```
Line 1
  continued...
Line 2
```

The first call to ReadContinuedLine will return "Line 1 continued..." and the second will return "Line 2".

Empty lines are never continued.

func (*Reader) ReadContinuedLineBytes

```
func (r *Reader) ReadContinuedLineBytes() ([]byte, error)
```

ReadContinuedLineBytes is like ReadContinuedLine but returns a []byte instead of a string.

func (*Reader) ReadDotBytes

```
func (r *Reader) ReadDotBytes() ([]byte, error)
```

ReadDotBytes reads a dot-encoding and returns the decoded data.

See the documentation for the DotReader method for details about dot-encoding.

func (*Reader) ReadDotLines

```
func (r *Reader) ReadDotLines() ([]string, error)
```

ReadDotLines reads a dot-encoding and returns a slice containing the decoded lines, with the final \r\n or \n elided from each.

See the documentation for the DotReader method for details about dot-encoding.

func (*Reader) ReadLine

```
func (r *Reader) ReadLine() (string, error)
```

ReadLine reads a single line from r, eliding the final \n or \r\n from the returned string.

func (*Reader) ReadLineBytes

```
func (r *Reader) ReadLineBytes() ([]byte, error)
```

ReadLineBytes is like ReadLine but returns a []byte instead of a string.

func (*Reader) ReadMIMEHeader

```
func (r *Reader) ReadMIMEHeader() (MIMEHeader, error)
```

ReadMIMEHeader reads a MIME-style header from r. The header is a sequence of possibly continued Key: Value lines ending in a blank line. The returned map m maps CanonicalMIMEHeaderKey(key) to a sequence of values in the same order encountered in the input.

For example, consider this input:

```
My-Key: Value 1
Long-Key: Even
       Longer Value
My-Key: Value 2
```

Given that input, ReadMIMEHeader returns the map:

```
map[string][]string{
    "My-Key": {"Value 1", "Value 2"},
    "Long-Key": {"Even Longer Value"},
}
```

func (*Reader) ReadResponse

```
func (r *Reader) ReadResponse(expectCode int) (code int, message string, err error)
```

ReadResponse reads a multi-line response of the form:

```
code-message line 1
code-message line 2
...
code message line n
```

where code is a three-digit status code. The first line starts with the code and a hyphen. The response is terminated by a line that starts with the same code followed by a space. Each line in message is separated by a newline (\n).

See page 36 of RFC 959 (https://www.ietf.org/rfc/rfc959.txt) for details of another form of response accepted:

```
code-message line 1
message line 2
...
code message line n
```

If the prefix of the status does not match the digits in expectCode, ReadResponse returns with err set to &Error{code, message}. For example, if expectCode is 31, an error will be returned if the status is not in the range [310,319].

An expectCode <= 0 disables the check of the status code.

type Writer

```
type Writer struct {
    W *bufio.Writer
    // contains filtered or unexported fields
}
```

A Writer implements convenience methods for writing requests or responses to a text protocol network connection.

func NewWriter

```
func NewWriter(w *bufio.Writer) *Writer
```

NewWriter returns a new Writer writing to w.

func (*Writer) DotWriter

```
func (w *Writer) DotWriter() io.WriteCloser
```

DotWriter returns a writer that can be used to write a dot-encoding to w. It takes care of inserting leading dots when necessary, translating line-ending \n into \r\n, and adding the final .\r\n line when the DotWriter is closed. The caller should close the DotWriter before the next call to a method on w.

See the documentation for Reader's DotReader method for details about dot-encoding.

func (*Writer) PrintfLine

```
func (w *Writer) PrintfLine(format string, args ...any) error
```

PrintfLine writes the formatted output followed by \r\n.

Source Files

View all

header.go
reader.go
writer.go
pipeline.go
textproto.go