Package http go1.15.2 Latest

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

Overview

Package http provides HTTP client and server implementations.

Get, Head, Post, and PostForm make HTTP (or HTTPS) requests:

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form",
    url.Values{"key": {"Value"}, "id": {"123"}})
```

The client must close the response body when finished with it:

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

For control over HTTP client headers, redirect policy, and other settings, create a Client:

```
client := &http.Client{
    CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
// ...

req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

For control over proxies, TLS configuration, keep-alives, compression, and other settings, create a Transport:

```
tr := &http.Transport{
    MaxIdleConns:       10,
    IdleConnTimeout:    30 * time.Second,
    DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

Clients and Transports are safe for concurrent use by multiple goroutines and for efficiency should only be created once and re-used.

ListenAndServe starts an HTTP server with a given address and handler. The handler is usually nil, which means to use DefaultServeMux. Handle and HandleFunc add handlers to DefaultServeMux:

```
http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

More control over the server's behavior is available by creating a custom Server:

```
s := &http.Server{
    Addr:           ":8080",
    Handler:        myHandler,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

Starting with Go 1.6, the http package has transparent support for the HTTP/2 protocol when using HTTPS. Programs that must disable HTTP/2 can do so by setting Transport.TLSNextProto (for clients) or Server.TLSNextProto (for servers) to a non-nil, empty map. Alternatively, the following GODEBUG environment variables are currently supported:

```
GODEBUG=http2client=0  # disable HTTP/2 client support
GODEBUG=http2server=0  # disable HTTP/2 server support
GODEBUG=http2debug=1   # enable verbose HTTP/2 debug logs
GODEBUG=http2debug=2   # ... even more verbose, with frame dumps
```

The GODEBUG variables are not covered by Go's API compatibility promise. Please report any issues before disabling HTTP/2 support: https://golang.org/s/http2bug

The http package's Transport and Server both automatically enable HTTP/2 support for simple configurations. To enable HTTP/2 for more complex configurations, to use lower-level HTTP/2

features, or to use a newer version of Go's http2 package, import "golang.org/x/net/http2" directly and use its ConfigureTransport and/or ConfigureServer functions. Manually configuring HTTP/2 via the golang.org/x/net/http2 package takes precedence over the net/http package's built-in HTTP/2 support.

Constants

```
const (
    MethodGet     = "GET"
    MethodHead    = "HEAD"
    MethodPost    = "POST"
    MethodPut     = "PUT"
    MethodPatch   = "PATCH" // RFC 5789
    MethodDelete  = "DELETE"
    MethodConnect = "CONNECT"
    MethodOptions = "OPTIONS"
    MethodTrace   = "TRACE"
)
```

Common HTTP methods.

Unless otherwise noted, these are defined in RFC 7231 section 4.3.

```
const (
    StatusContinue           = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing         = 102 // RFC 2518, 10.1
    StatusEarlyHints         = 103 // RFC 8297

    StatusOK                   = 200 // RFC 7231, 6.3.1
    StatusCreated              = 201 // RFC 7231, 6.3.2
    StatusAccepted             = 202 // RFC 7231, 6.3.3
    StatusNonAuthoritativeInfo = 203 // RFC 7231, 6.3.4
    StatusNoContent            = 204 // RFC 7231, 6.3.5
    StatusResetContent         = 205 // RFC 7231, 6.3.6
    StatusPartialContent       = 206 // RFC 7233, 4.1
    StatusMultiStatus          = 207 // RFC 4918, 11.1
    StatusAlreadyReported      = 208 // RFC 5842, 7.1
    StatusIMUsed               = 226 // RFC 3229, 10.4.1

    StatusMultipleChoices  = 300 // RFC 7231, 6.4.1
    StatusMovedPermanently = 301 // RFC 7231, 6.4.2
    StatusFound            = 302 // RFC 7231, 6.4.3
    StatusSeeOther         = 303 // RFC 7231, 6.4.4
    StatusNotModified      = 304 // RFC 7232, 4.1
    StatusUseProxy         = 305 // RFC 7231, 6.4.5

    StatusTemporaryRedirect = 307 // RFC 7231, 6.4.7
    StatusPermanentRedirect = 308 // RFC 7538, 3

    StatusBadRequest                   = 400 // RFC 7231, 6.5.1
    StatusUnauthorized                 = 401 // RFC 7235, 3.1
```

```
    StatusPaymentRequired               = 402 // RFC 7231, 6.5.2
    StatusForbidden                     = 403 // RFC 7231, 6.5.3
    StatusNotFound                      = 404 // RFC 7231, 6.5.4
    StatusMethodNotAllowed              = 405 // RFC 7231, 6.5.5
    StatusNotAcceptable                 = 406 // RFC 7231, 6.5.6
    StatusProxyAuthRequired             = 407 // RFC 7235, 3.2
    StatusRequestTimeout                = 408 // RFC 7231, 6.5.7
    StatusConflict                      = 409 // RFC 7231, 6.5.8
    StatusGone                          = 410 // RFC 7231, 6.5.9
    StatusLengthRequired                = 411 // RFC 7231, 6.5.10
    StatusPreconditionFailed            = 412 // RFC 7232, 4.2
    StatusRequestEntityTooLarge         = 413 // RFC 7231, 6.5.11
    StatusRequestURITooLong             = 414 // RFC 7231, 6.5.12
    StatusUnsupportedMediaType          = 415 // RFC 7231, 6.5.13
    StatusRequestedRangeNotSatisfiable = 416 // RFC 7233, 4.4
    StatusExpectationFailed             = 417 // RFC 7231, 6.5.14
    StatusTeapot                        = 418 // RFC 7168, 2.3.3
    StatusMisdirectedRequest            = 421 // RFC 7540, 9.1.2
    StatusUnprocessableEntity           = 422 // RFC 4918, 11.2
    StatusLocked                        = 423 // RFC 4918, 11.3
    StatusFailedDependency              = 424 // RFC 4918, 11.4
    StatusTooEarly                      = 425 // RFC 8470, 5.2.
    StatusUpgradeRequired               = 426 // RFC 7231, 6.5.15
    StatusPreconditionRequired          = 428 // RFC 6585, 3
    StatusTooManyRequests               = 429 // RFC 6585, 4
    StatusRequestHeaderFieldsTooLarge   = 431 // RFC 6585, 5
    StatusUnavailableForLegalReasons    = 451 // RFC 7725, 3

    StatusInternalServerError           = 500 // RFC 7231, 6.6.1
    StatusNotImplemented                = 501 // RFC 7231, 6.6.2
    StatusBadGateway                    = 502 // RFC 7231, 6.6.3
    StatusServiceUnavailable            = 503 // RFC 7231, 6.6.4
    StatusGatewayTimeout                = 504 // RFC 7231, 6.6.5
    StatusHTTPVersionNotSupported       = 505 // RFC 7231, 6.6.6
    StatusVariantAlsoNegotiates         = 506 // RFC 2295, 8.1
    StatusInsufficientStorage           = 507 // RFC 4918, 11.5
    StatusLoopDetected                  = 508 // RFC 5842, 7.2
    StatusNotExtended                   = 510 // RFC 2774, 7
    StatusNetworkAuthenticationRequired = 511 // RFC 6585, 6
)
```

HTTP status codes as registered with IANA. See: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

```
const DefaultMaxHeaderBytes = 1 << 20 // 1 MB
```

DefaultMaxHeaderBytes is the maximum permitted size of the headers in an HTTP request. This can be overridden by setting Server.MaxHeaderBytes.

```
const DefaultMaxIdleConnsPerHost = 2
```

DefaultMaxIdleConnsPerHost is the default value of Transport's MaxIdleConnsPerHost.

```
const TimeFormat = "Mon, 02 Jan 2006 15:04:05 GMT"
```

TimeFormat is the time format to use when generating times in HTTP headers. It is like time.RFC1123 but hard-codes GMT as the time zone. The time being formatted must be in UTC for Format to generate the correct format.

For parsing this time format, see ParseTime.

```
const TrailerPrefix = "Trailer:"
```

TrailerPrefix is a magic prefix for ResponseWriter.Header map keys that, if present, signals that the map entry is actually for the response trailers, and not the response headers. The prefix is stripped after the ServeHTTP call finishes and the values are sent in the trailers.

This mechanism is intended only for trailers that are not known prior to the headers being written. If the set of trailers is fixed or known before the header is written, the normal Go trailers mechanism is preferred:

```
https://golang.org/pkg/net/http/#ResponseWriter
https://golang.org/pkg/net/http/#example_ResponseWriter_trailers
```

Variables

```
var (
    // ErrNotSupported is returned by the Push method of Pusher
    // implementations to indicate that HTTP/2 Push support is not
    // available.
    ErrNotSupported = &ProtocolError{"feature not supported"}

    // Deprecated: ErrUnexpectedTrailer is no longer returned by
    // anything in the net/http package. Callers should not
    // compare errors against this variable.
    ErrUnexpectedTrailer = &ProtocolError{"trailer header without chunked transfer en

    // ErrMissingBoundary is returned by Request.MultipartReader when the
    // request's Content-Type does not include a "boundary" parameter.
    ErrMissingBoundary = &ProtocolError{"no multipart boundary param in Content-Type"

    // ErrNotMultipart is returned by Request.MultipartReader when the
    // request's Content-Type is not multipart/form-data.
    ErrNotMultipart = &ProtocolError{"request Content-Type isn't multipart/form-data"

    // Deprecated: ErrHeaderTooLong is no longer returned by
    // anything in the net/http package. Callers should not
    // compare errors against this variable.
    ErrHeaderTooLong = &ProtocolError{"header too long"}
```

```
    // Deprecated: ErrShortBody is no longer returned by
    // anything in the net/http package. Callers should not
    // compare errors against this variable.
    ErrShortBody = &ProtocolError{"entity body too short"}

    // Deprecated: ErrMissingContentLength is no longer returned by
    // anything in the net/http package. Callers should not
    // compare errors against this variable.
    ErrMissingContentLength = &ProtocolError{"missing ContentLength in HEAD response"
)
```

```
var (
    // ErrBodyNotAllowed is returned by ResponseWriter.Write calls
    // when the HTTP method or response code does not permit a
    // body.
    ErrBodyNotAllowed = errors.New("http: request method or response status code does

    // ErrHijacked is returned by ResponseWriter.Write calls when
    // the underlying connection has been hijacked using the
    // Hijacker interface. A zero-byte write on a hijacked
    // connection will return ErrHijacked without any other side
    // effects.
    ErrHijacked = errors.New("http: connection has been hijacked")

    // ErrContentLength is returned by ResponseWriter.Write calls
    // when a Handler set a Content-Length response header with a
    // declared size and then attempted to write more bytes than
    // declared.
    ErrContentLength = errors.New("http: wrote more than the declared Content-Length"

    // Deprecated: ErrWriteAfterFlush is no longer returned by
    // anything in the net/http package. Callers should not
    // compare errors against this variable.
    ErrWriteAfterFlush = errors.New("unused")
)
```

Errors used by the HTTP server.

```
var (
    // ServerContextKey is a context key. It can be used in HTTP
    // handlers with Context.Value to access the server that
    // started the handler. The associated value will be of
    // type *Server.
    ServerContextKey = &contextKey{"http-server"}

    // LocalAddrContextKey is a context key. It can be used in
    // HTTP handlers with Context.Value to access the local
    // address the connection arrived on.
    // The associated value will be of type net.Addr.
    LocalAddrContextKey = &contextKey{"local-addr"}
)
```

```
var DefaultClient = &Client{}
```

DefaultClient is the default Client and is used by Get, Head, and Post.

```
var DefaultServeMux = &defaultServeMux
```

DefaultServeMux is the default ServeMux used by Serve.

```
var ErrAbortHandler = errors.New("net/http: abort Handler")
```

ErrAbortHandler is a sentinel panic value to abort a handler. While any panic from ServeHTTP aborts the response to the client, panicking with ErrAbortHandler also suppresses logging of a stack trace to the server's error log.

```
var ErrBodyReadAfterClose = errors.New("http: invalid Read on closed Body")
```

ErrBodyReadAfterClose is returned when reading a Request or Response Body after the body has been closed. This typically happens when the body is read after an HTTP Handler calls WriteHeader or Write on its ResponseWriter.

```
var ErrHandlerTimeout = errors.New("http: Handler timeout")
```

ErrHandlerTimeout is returned on ResponseWriter Write calls in handlers which have timed out.

```
var ErrLineTooLong = internal.ErrLineTooLong
```

ErrLineTooLong is returned when reading request or response bodies with malformed chunked encoding.

```
var ErrMissingFile = errors.New("http: no such file")
```

ErrMissingFile is returned by FormFile when the provided file field name is either not present in the request or not a file field.

```
var ErrNoCookie = errors.New("http: named cookie not present")
```

ErrNoCookie is returned by Request's Cookie method when a cookie is not found.

```
var ErrNoLocation = errors.New("http: no Location header in response")
```

ErrNoLocation is returned by Response's Location method when no Location header is present.

```
var ErrServerClosed = errors.New("http: Server closed")
```

ErrServerClosed is returned by the Server's Serve, ServeTLS, ListenAndServe, and ListenAndServeTLS methods after a call to Shutdown or Close.

```
var ErrSkipAltProtocol = errors.New("net/http: skip alternate protocol")
```

ErrSkipAltProtocol is a sentinel error value defined by Transport.RegisterProtocol.

```
var ErrUseLastResponse = errors.New("net/http: use last response")
```

ErrUseLastResponse can be returned by Client.CheckRedirect hooks to control how redirects are processed. If returned, the next request is not sent and the most recent response is returned with its body unclosed.

```
var NoBody = noBody{}
```

NoBody is an io.ReadCloser with no bytes. Read always returns EOF and Close always returns nil. It can be used in an outgoing client request to explicitly signal that a request has zero bytes. An alternative, however, is to simply set Request.Body to nil.

func CanonicalHeaderKey

```
func CanonicalHeaderKey(s string) string
```

CanonicalHeaderKey returns the canonical format of the 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". If s contains a space or invalid header field bytes, it is returned without modifications.

func DetectContentType

```
func DetectContentType(data []byte) string
```

DetectContentType implements the algorithm described at https://mimesniff.spec.whatwg.org/ to determine the Content-Type of the given data. It considers at most the first 512 bytes of data. DetectContentType always returns a valid MIME type: if it cannot determine a more specific one, it returns "application/octet-stream".

func Error

```
func Error(w ResponseWriter, error string, code int)
```

Error replies to the request with the specified error message and HTTP code. It does not otherwise end the request; the caller should ensure no further writes are done to w. The error message should be plain text.

func Handle

```
func Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

func HandleFunc

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc registers the handler function for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

func ListenAndServe

```
func ListenAndServe(addr string, handler Handler) error
```

ListenAndServe listens on the TCP network address addr and then calls Serve with handler to handle requests on incoming connections. Accepted connections are configured to enable TCP keep-alives.

The handler is typically nil, in which case the DefaultServeMux is used.

ListenAndServe always returns a non-nil error.

func ListenAndServeTLS

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

ListenAndServeTLS acts identically to ListenAndServe, except that it expects HTTPS connections. Additionally, files containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

func MaxBytesReader

```
func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.ReadCloser
```

MaxBytesReader is similar to io.LimitReader but is intended for limiting the size of incoming request bodies. In contrast to io.LimitReader, MaxBytesReader's result is a ReadCloser, returns a non-EOF error for a Read beyond the limit, and closes the underlying reader when its Close method is called.

MaxBytesReader prevents clients from accidentally or maliciously sending a large request and wasting server resources.

func NotFound

```
func NotFound(w ResponseWriter, r *Request)
```

NotFound replies to the request with an HTTP 404 not found error.

func ParseHTTPVersion

```
func ParseHTTPVersion(vers string) (major, minor int, ok bool)
```

ParseHTTPVersion parses an HTTP version string. "HTTP/1.0" returns (1, 0, true).

func ParseTime

```
func ParseTime(text string) (t time.Time, err error)
```

ParseTime parses a time header (such as the Date: header), trying each of the three formats allowed by HTTP/1.1: TimeFormat, time.RFC850, and time.ANSIC.

func ProxyFromEnvironment

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

ProxyFromEnvironment returns the URL of the proxy to use for a given request, as indicated by the environment variables HTTP_PROXY, HTTPS_PROXY and NO_PROXY (or the lowercase versions thereof). HTTPS_PROXY takes precedence over HTTP_PROXY for https requests.

The environment values may be either a complete URL or a "host[:port]", in which case the "http" scheme is assumed. An error is returned if the value is a different form.

A nil URL and nil error are returned if no proxy is defined in the environment, or a proxy should not be used for the given request, as defined by NO_PROXY.

As a special case, if req.URL.Host is "localhost" (with or without a port number), then a nil URL and nil error will be returned.

func ProxyURL

```
func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
```

ProxyURL returns a proxy function (for use in a Transport) that always returns the same URL.

func Redirect

```
func Redirect(w ResponseWriter, r *Request, url string, code int)
```

Redirect replies to the request with a redirect to url, which may be a path relative to the request path.

The provided code should be in the 3xx range and is usually StatusMovedPermanently, StatusFound or StatusSeeOther.

If the Content-Type header has not been set, Redirect sets it to "text/html; charset=utf-8" and writes a small HTML body. Setting the Content-Type header to any value, including nil, disables that behavior.

func Serve

```
func Serve(l net.Listener, handler Handler) error
```

Serve accepts incoming HTTP connections on the listener l, creating a new service goroutine for each. The service goroutines read requests and then call handler to reply to them.

The handler is typically nil, in which case the DefaultServeMux is used.

HTTP/2 support is only enabled if the Listener returns *tls.Conn connections and they were configured with "h2" in the TLS Config.NextProtos.

Serve always returns a non-nil error.

func ServeContent

```
func ServeContent(w ResponseWriter, req *Request, name string, modtime time.Time, con
```

ServeContent replies to the request using the content in the provided ReadSeeker. The main benefit of ServeContent over io.Copy is that it handles Range requests properly, sets the MIME type, and handles If-Match, If-Unmodified-Since, If-None-Match, If-Modified-Since, and If-Range requests.

If the response's Content-Type header is not set, ServeContent first tries to deduce the type from name's file extension and, if that fails, falls back to reading the first block of the content and passing it to DetectContentType. The name is otherwise unused; in particular it can be empty and is never sent in the response.

If modtime is not the zero time or Unix epoch, ServeContent includes it in a Last-Modified header in the response. If the request includes an If-Modified-Since header, ServeContent uses modtime to decide whether the content needs to be sent at all.

The content's Seek method must work: ServeContent uses a seek to the end of the content to determine its size.

If the caller has set w's ETag header formatted per RFC 7232, section 2.3, ServeContent uses it to handle requests using If-Match, If-None-Match, or If-Range.

Note that *os.File implements the io.ReadSeeker interface.

func ServeFile

```
func ServeFile(w ResponseWriter, r *Request, name string)
```

ServeFile replies to the request with the contents of the named file or directory.

If the provided file or directory name is a relative path, it is interpreted relative to the current directory and may ascend to parent directories. If the provided name is constructed from user input, it should be sanitized before calling ServeFile.

As a precaution, ServeFile will reject requests where r.URL.Path contains a ".." path element; this protects against callers who might unsafely use filepath.Join on r.URL.Path without sanitizing it and then use that filepath.Join result as the name argument.

As another special case, ServeFile redirects any request where r.URL.Path ends in "/index.html" to the same path, without the final "index.html". To avoid such redirects either modify the path or use ServeContent.

Outside of those two special cases, ServeFile does not use r.URL.Path for selecting the file or directory to serve; only the file or directory provided in the name argument is used.

func ServeTLS

```
func ServeTLS(l net.Listener, handler Handler, certFile, keyFile string) error
```

ServeTLS accepts incoming HTTPS connections on the listener l, creating a new service goroutine for each. The service goroutines read requests and then call handler to reply to them.

The handler is typically nil, in which case the DefaultServeMux is used.

Additionally, files containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

ServeTLS always returns a non-nil error.

func SetCookie

```
func SetCookie(w ResponseWriter, cookie *Cookie)
```

SetCookie adds a Set-Cookie header to the provided ResponseWriter's headers. The provided cookie must have a valid Name. Invalid cookies may be silently dropped.

func StatusText

```
func StatusText(code int) string
```

StatusText returns a text for the HTTP status code. It returns the empty string if the code is unknown.

type Client

```
type Client struct {
    // Transport specifies the mechanism by which individual
    // HTTP requests are made.
    // If nil, DefaultTransport is used.
    Transport RoundTripper

    // CheckRedirect specifies the policy for handling redirects.
    // If CheckRedirect is not nil, the client calls it before
    // following an HTTP redirect. The arguments req and via are
    // the upcoming request and the requests made already, oldest
    // first. If CheckRedirect returns an error, the Client's Get
    // method returns both the previous Response (with its Body
    // closed) and CheckRedirect's error (wrapped in a url.Error)
    // instead of issuing the Request req.
    // As a special case, if CheckRedirect returns ErrUseLastResponse,
    // then the most recent response is returned with its body
    // unclosed, along with a nil error.
    //
    // If CheckRedirect is nil, the Client uses its default policy,
    // which is to stop after 10 consecutive requests.
    CheckRedirect func(req *Request, via []*Request) error

    // Jar specifies the cookie jar.
    //
    // The Jar is used to insert relevant cookies into every
    // outbound Request and is updated with the cookie values
    // of every inbound Response. The Jar is consulted for every
    // redirect that the Client follows.
    //
    // If Jar is nil, cookies are only sent if they are explicitly
    // set on the Request.
    Jar CookieJar

    // Timeout specifies a time limit for requests made by this
    // Client. The timeout includes connection time, any
    // redirects, and reading the response body. The timer remains
    // running after Get, Head, Post, or Do return and will
    // interrupt reading of the Response.Body.
    //
    // A Timeout of zero means no timeout.
    //
    // The Client cancels requests to the underlying Transport
    // as if the Request's Context ended.
    //
    // For compatibility, the Client will also use the deprecated
    // CancelRequest method on Transport if found. New
    // RoundTripper implementations should use the Request's Context
    // for cancellation instead of implementing CancelRequest.
    Timeout time.Duration
}
```

A Client is an HTTP client. Its zero value (DefaultClient) is a usable client that uses DefaultTransport.

The Client's Transport typically has internal state (cached TCP connections), so Clients should be reused instead of created as needed. Clients are safe for concurrent use by multiple goroutines.

A Client is higher-level than a RoundTripper (such as Transport) and additionally handles HTTP details such as cookies and redirects.

When following redirects, the Client will forward all headers set on the initial Request except:

• when forwarding sensitive headers like "Authorization", "WWW-Authenticate", and "Cookie" to untrusted targets. These headers will be ignored when following a redirect to a domain that is not a subdomain match or exact match of the initial domain. For example, a redirect from "foo.com" to either "foo.com" or "sub.foo.com" will forward the sensitive headers, but a redirect to "bar.com" will not.

• when forwarding the "Cookie" header with a non-nil cookie Jar. Since each redirect may mutate the state of the cookie jar, a redirect may possibly alter a cookie set in the initial request. When forwarding the "Cookie" header, any mutated cookies will be omitted, with the expectation that the Jar will insert those mutated cookies with the updated values (assuming the origin matches). If Jar is nil, the initial cookies are forwarded without change.

func (*Client) CloseIdleConnections

```
func (c *Client) CloseIdleConnections()
```

CloseIdleConnections closes any connections on its Transport which were previously connected from previous requests but are now sitting idle in a "keep-alive" state. It does not interrupt any connections currently in use.

If the Client's Transport does not have a CloseIdleConnections method then this method does nothing.

func (*Client) Do

```
func (c *Client) Do(req *Request) (*Response, error)
```

Do sends an HTTP request and returns an HTTP response, following policy (such as redirects, cookies, auth) as configured on the client.

An error is returned if caused by client policy (such as CheckRedirect), or failure to speak HTTP (such as a network connectivity problem). A non-2xx status code doesn't cause an error.

If the returned error is nil, the Response will contain a non-nil Body which the user is expected to close. If the Body is not both read to EOF and closed, the Client's underlying RoundTripper (typically Transport) may not be able to re-use a persistent TCP connection to the server for a subsequent "keep-alive" request.

The request Body, if non-nil, will be closed by the underlying Transport, even on errors.

On error, any Response can be ignored. A non-nil Response with a non-nil error only occurs when CheckRedirect fails, and even then the returned Response.Body is already closed.

Generally Get, Post, or PostForm will be used instead of Do.

If the server replies with a redirect, the Client first uses the CheckRedirect function to determine whether the redirect should be followed. If permitted, a 301, 302, or 303 redirect causes subsequent requests to use HTTP method GET (or HEAD if the original request was HEAD), with no body. A 307 or 308 redirect preserves the original HTTP method and body, provided that the Request.GetBody function is defined. The NewRequest function automatically sets GetBody for common standard library body types.

Any returned error will be of type *url.Error. The url.Error value's Timeout method will report true if request timed out or was canceled.

func (*Client) Get

```
func (c *Client) Get(url string) (resp *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect after calling the Client's CheckRedirect function:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)
```

An error is returned if the Client's CheckRedirect function fails or if there was an HTTP protocol error. A non-2xx response doesn't cause an error. Any returned error will be of type *url.Error. The url.Error value's Timeout method will report true if the request timed out.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

To make a request with custom headers, use NewRequest and Client.Do.

func (*Client) Head

```
func (c *Client) Head(url string) (resp *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the following redirect codes, Head follows the redirect after calling the Client's CheckRedirect function:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
```

```
307 (Temporary Redirect)
308 (Permanent Redirect)
```

func (*Client) Post

```
func (c *Client) Post(url, contentType string, body io.Reader) (resp *Response, err e
```

Post issues a POST to the specified URL.

Caller should close resp.Body when done reading from it.

If the provided body is an io.Closer, it is closed after the request.

To set custom headers, use NewRequest and Client.Do.

See the Client.Do method documentation for details on how redirects are handled.

func (*Client) PostForm

```
func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)
```

PostForm issues a POST to the specified URL, with data's keys and values URL-encoded as the request body.

The Content-Type header is set to application/x-www-form-urlencoded. To set other headers, use NewRequest and Client.Do.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

See the Client.Do method documentation for details on how redirects are handled.

type CloseNotifier

```
type CloseNotifier interface {
    // CloseNotify returns a channel that receives at most a
    // single value (true) when the client connection has gone
    // away.
    //
    // CloseNotify may wait to notify until Request.Body has been
    // fully read.
    //
    // After the Handler has returned, there is no guarantee
    // that the channel receives a value.
    //
    // If the protocol is HTTP/1.1 and CloseNotify is called while
    // processing an idempotent request (such a GET) while
    // HTTP/1.1 pipelining is in use, the arrival of a subsequent
    // pipelined request may cause a value to be sent on the
    // returned channel. In practice HTTP/1.1 pipelining is not
```

```
    // enabled in browsers and not seen often in the wild. If this
    // is a problem, use HTTP/2 or only use CloseNotify on methods
    // such as POST.
    CloseNotify() <-chan bool
}
```

The CloseNotifier interface is implemented by ResponseWriters which allow detecting when the underlying connection has gone away.

This mechanism can be used to cancel long operations on the server if the client has disconnected before the response is ready.

Deprecated: the CloseNotifier interface predates Go's context package. New code should use Request.Context instead.

type ConnState

```
type ConnState int
```

A ConnState represents the state of a client connection to a server. It's used by the optional Server.ConnState hook.

```
const (
    // StateNew represents a new connection that is expected to
    // send a request immediately. Connections begin at this
    // state and then transition to either StateActive or
    // StateClosed.
    StateNew ConnState = iota

    // StateActive represents a connection that has read 1 or more
    // bytes of a request. The Server.ConnState hook for
    // StateActive fires before the request has entered a handler
    // and doesn't fire again until the request has been
    // handled. After the request is handled, the state
    // transitions to StateClosed, StateHijacked, or StateIdle.
    // For HTTP/2, StateActive fires on the transition from zero
    // to one active request, and only transitions away once all
    // active requests are complete. That means that ConnState
    // cannot be used to do per-request work; ConnState only notes
    // the overall state of the connection.
    StateActive

    // StateIdle represents a connection that has finished
    // handling a request and is in the keep-alive state, waiting
    // for a new request. Connections transition from StateIdle
    // to either StateActive or StateClosed.
    StateIdle

    // StateHijacked represents a hijacked connection.
    // This is a terminal state. It does not transition to StateClosed.
    StateHijacked
```

```
    // StateClosed represents a closed connection.
    // This is a terminal state. Hijacked connections do not
    // transition to StateClosed.
    StateClosed
)
```

func (ConnState) String

```
func (c ConnState) String() string
```

type Cookie

```
type Cookie struct {
    Name  string
    Value string

    Path       string    // optional
    Domain     string    // optional
    Expires    time.Time // optional
    RawExpires string    // for reading cookies only

    // MaxAge=0 means no 'Max-Age' attribute specified.
    // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
    // MaxAge>0 means Max-Age attribute present and given in seconds
    MaxAge   int
    Secure   bool
    HttpOnly bool
    SameSite SameSite
    Raw      string
    Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

A Cookie represents an HTTP cookie as sent in the Set-Cookie header of an HTTP response or the Cookie header of an HTTP request.

See https://tools.ietf.org/html/rfc6265 for details.

func (*Cookie) String

```
func (c *Cookie) String() string
```

String returns the serialization of the cookie for use in a Cookie header (if only Name and Value are set) or a Set-Cookie response header (if other fields are set). If c is nil or c.Name is invalid, the empty string is returned.

type CookieJar

```
type CookieJar interface {
    // SetCookies handles the receipt of the cookies in a reply for the
    // given URL.  It may or may not choose to save the cookies, depending
    // on the jar's policy and implementation.
    SetCookies(u *url.URL, cookies []*Cookie)

    // Cookies returns the cookies to send in a request for the given URL.
    // It is up to the implementation to honor the standard cookie use
    // restrictions such as in RFC 6265.
    Cookies(u *url.URL) []*Cookie
}
```

A CookieJar manages storage and use of cookies in HTTP requests.

Implementations of CookieJar must be safe for concurrent use by multiple goroutines.

The net/http/cookiejar package provides a CookieJar implementation.

type Dir

```
type Dir string
```

A Dir implements FileSystem using the native file system restricted to a specific directory tree.

While the FileSystem.Open method takes '/'-separated paths, a Dir's string value is a filename on the native file system, not a URL, so it is separated by filepath.Separator, which isn't necessarily '/'.

Note that Dir could expose sensitive files and directories. Dir will follow symlinks pointing out of the directory tree, which can be especially dangerous if serving from a directory in which users are able to create arbitrary symlinks. Dir will also allow access to files and directories starting with a period, which could expose sensitive directories like .git or sensitive files like .htpasswd. To exclude files with a leading period, remove the files/directories from the server or create a custom FileSystem implementation.

An empty Dir is treated as ".".

func (Dir) Open

```
func (d Dir) Open(name string) (File, error)
```

Open implements FileSystem using os.Open, opening files for reading rooted and relative to the directory d.

type File

```
type File interface {
    io.Closer
    io.Reader
    io.Seeker
```

```
    Readdir(count int) ([]os.FileInfo, error)
    Stat() (os.FileInfo, error)
}
```

A File is returned by a FileSystem's Open method and can be served by the FileServer implementation.

The methods should behave the same as those on an *os.File.

type FileSystem

```
type FileSystem interface {
    Open(name string) (File, error)
}
```

A FileSystem implements access to a collection of named files. The elements in a file path are separated by slash ('/', U+002F) characters, regardless of host operating system convention.

type Flusher

```
type Flusher interface {
    // Flush sends any buffered data to the client.
    Flush()
}
```

The Flusher interface is implemented by ResponseWriters that allow an HTTP handler to flush buffered data to the client.

The default HTTP/1.x and HTTP/2 ResponseWriter implementations support Flusher, but ResponseWriter wrappers may not. Handlers should always test for this ability at runtime.

Note that even for ResponseWriters that support Flush, if the client is connected through an HTTP proxy, the buffered data may not reach the client until the response completes.

type Handler

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

A Handler responds to an HTTP request.

ServeHTTP should write reply headers and data to the ResponseWriter and then return. Returning signals that the request is finished; it is not valid to use the ResponseWriter or read from the Request.Body after or concurrently with the completion of the ServeHTTP call.

Depending on the HTTP client software, HTTP protocol version, and any intermediaries between the client and the Go server, it may not be possible to read from the Request.Body after writing to the ResponseWriter. Cautious handlers should read the Request.Body first, and then reply.

Except for reading the body, handlers should not modify the provided Request.

If ServeHTTP panics, the server (the caller of ServeHTTP) assumes that the effect of the panic was isolated to the active request. It recovers the panic, logs a stack trace to the server error log, and either closes the network connection or sends an HTTP/2 RST_STREAM, depending on the HTTP protocol. To abort a handler so the client sees an interrupted response but the server doesn't log an error, panic with the value ErrAbortHandler.

func FileServer

```
func FileServer(root FileSystem) Handler
```

FileServer returns a handler that serves HTTP requests with the contents of the file system rooted at root.

To use the operating system's file system implementation, use http.Dir:

```
http.Handle("/", http.FileServer(http.Dir("/tmp")))
```

As a special case, the returned file server redirects any request ending in "/index.html" to the same path, without the final "index.html".

func NotFoundHandler

```
func NotFoundHandler() Handler
```

NotFoundHandler returns a simple request handler that replies to each request with a ``404 page not found" reply.

func RedirectHandler

```
func RedirectHandler(url string, code int) Handler
```

RedirectHandler returns a request handler that redirects each request it receives to the given url using the given status code.

The provided code should be in the 3xx range and is usually StatusMovedPermanently, StatusFound or StatusSeeOther.

func StripPrefix

```
func StripPrefix(prefix string, h Handler) Handler
```

StripPrefix returns a handler that serves HTTP requests by removing the given prefix from the request URL's Path and invoking the handler h. StripPrefix handles a request for a path that doesn't begin with prefix by replying with an HTTP 404 not found error.

func TimeoutHandler

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

TimeoutHandler returns a Handler that runs h with the given time limit.

The new Handler calls h.ServeHTTP to handle each request, but if a call runs for longer than its time limit, the handler responds with a 503 Service Unavailable error and the given message in its body. (If msg is empty, a suitable default message will be sent.) After such a timeout, writes by h to its ResponseWriter will return ErrHandlerTimeout.

TimeoutHandler supports the Pusher interface but does not support the Hijacker or Flusher interfaces.

type HandlerFunc

```
type HandlerFunc func(ResponseWriter, *Request)
```

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler that calls f.

func (HandlerFunc) ServeHTTP

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP calls f(w, r).

type Header

```
type Header map[string][]string
```

A Header represents the key-value pairs in an HTTP header.

The keys should be in canonical form, as returned by CanonicalHeaderKey.

func (Header) Add

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

Add adds the key, value pair to the header. It appends to any existing values associated with key. The key is case insensitive; it is canonicalized by CanonicalHeaderKey.

func (Header) Clone

```
func (h Header) Clone() Header
```

Clone returns a copy of h or nil if h is nil.

func (Header) Del

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

Del deletes the values associated with key. The key is case insensitive; it is canonicalized by CanonicalHeaderKey.

func (Header) Get

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

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

func (Header) Set

```
func (h Header) 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. The key is case insensitive; it is canonicalized by textproto.CanonicalMIMEHeaderKey. To use non-canonical keys, assign to the map directly.

func (Header) Values

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

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

func (Header) Write

```
func (h Header) Write(w io.Writer) error
```

Write writes a header in wire format.

func (Header) WriteSubset

```
func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) error
```

WriteSubset writes a header in wire format. If exclude is not nil, keys where exclude[key] == true are not written. Keys are not canonicalized before checking the exclude map.

type Hijacker

```
type Hijacker interface {
    // Hijack lets the caller take over the connection.
    // After a call to Hijack the HTTP server library
    // will not do anything else with the connection.
    //
    // It becomes the caller's responsibility to manage
    // and close the connection.
    //
    // The returned net.Conn may have read or write deadlines
    // already set, depending on the configuration of the
    // Server. It is the caller's responsibility to set
    // or clear those deadlines as needed.
    //
    // The returned bufio.Reader may contain unprocessed buffered
    // data from the client.
    //
    // After a call to Hijack, the original Request.Body must not
    // be used. The original Request's Context remains valid and
    // is not canceled until the Request's ServeHTTP method
    // returns.
    Hijack() (net.Conn, *bufio.ReadWriter, error)
}
```

The Hijacker interface is implemented by ResponseWriters that allow an HTTP handler to take over the connection.

The default ResponseWriter for HTTP/1.x connections supports Hijacker, but HTTP/2 connections intentionally do not. ResponseWriter wrappers may also not support Hijacker. Handlers should always test for this ability at runtime.

type ProtocolError

```
type ProtocolError struct {
    ErrorString string
}
```

ProtocolError represents an HTTP protocol error.

Deprecated: Not all errors in the http package related to protocol errors are of type ProtocolError.

func (*ProtocolError) Error

```
func (pe *ProtocolError) Error() string
```

type PushOptions

```
type PushOptions struct {
    // Method specifies the HTTP method for the promised request.
```

```
    // If set, it must be "GET" or "HEAD". Empty means "GET".
    Method string

    // Header specifies additional promised request headers. This cannot
    // include HTTP/2 pseudo header fields like ":path" and ":scheme",
    // which will be added automatically.
    Header Header
}
```

PushOptions describes options for Pusher.Push.

type Pusher

```
type Pusher interface {
    // Push initiates an HTTP/2 server push. This constructs a synthetic
    // request using the given target and options, serializes that request
    // into a PUSH_PROMISE frame, then dispatches that request using the
    // server's request handler. If opts is nil, default options are used.
    //
    // The target must either be an absolute path (like "/path") or an absolute
    // URL that contains a valid host and the same scheme as the parent request.
    // If the target is a path, it will inherit the scheme and host of the
    // parent request.
    //
    // The HTTP/2 spec disallows recursive pushes and cross-authority pushes.
    // Push may or may not detect these invalid pushes; however, invalid
    // pushes will be detected and canceled by conforming clients.
    //
    // Handlers that wish to push URL X should call Push before sending any
    // data that may trigger a request for URL X. This avoids a race where the
    // client issues requests for X before receiving the PUSH_PROMISE for X.
    //
    // Push will run in a separate goroutine making the order of arrival
    // non-deterministic. Any required synchronization needs to be implemented
    // by the caller.
    //
    // Push returns ErrNotSupported if the client has disabled push or if push
    // is not supported on the underlying connection.
    Push(target string, opts *PushOptions) error
}
```

Pusher is the interface implemented by ResponseWriters that support HTTP/2 server push. For more background, see https://tools.ietf.org/html/rfc7540#section-8.2.

type Request

```
type Request struct {
    // Method specifies the HTTP method (GET, POST, PUT, etc.).
    // For client requests, an empty string means GET.
    //
    // Go's HTTP client does not support sending a request with
```

```
	// the CONNECT method. See the documentation on Transport for
	// details.
	Method string

	// URL specifies either the URI being requested (for server
	// requests) or the URL to access (for client requests).
	//
	// For server requests, the URL is parsed from the URI
	// supplied on the Request-Line as stored in RequestURI.  For
	// most requests, fields other than Path and RawQuery will be
	// empty. (See RFC 7230, Section 5.3)
	//
	// For client requests, the URL's Host specifies the server to
	// connect to, while the Request's Host field optionally
	// specifies the Host header value to send in the HTTP
	// request.
	URL *url.URL

	// The protocol version for incoming server requests.
	//
	// For client requests, these fields are ignored. The HTTP
	// client code always uses either HTTP/1.1 or HTTP/2.
	// See the docs on Transport for details.
	Proto      string // "HTTP/1.0"
	ProtoMajor int    // 1
	ProtoMinor int    // 0

	// Header contains the request header fields either received
	// by the server or to be sent by the client.
	//
	// If a server received a request with header lines,
	//
	//  Host: example.com
	//  accept-encoding: gzip, deflate
	//  Accept-Language: en-us
	//  fOO: Bar
	//  foo: two
	//
	// then
	//
	//  Header = map[string][]string{
	//      "Accept-Encoding": {"gzip, deflate"},
	//      "Accept-Language": {"en-us"},
	//      "Foo": {"Bar", "two"},
	//  }
	//
	// For incoming requests, the Host header is promoted to the
	// Request.Host field and removed from the Header map.
	//
	// HTTP defines that header names are case-insensitive. The
	// request parser implements this by using CanonicalHeaderKey,
	// making the first character and any characters following a
	// hyphen uppercase and the rest lowercase.
	//
```

```
	// For client requests, certain headers such as Content-Length
	// and Connection are automatically written when needed and
	// values in Header may be ignored. See the documentation
	// for the Request.Write method.
	Header Header

	// Body is the request's body.
	//
	// For client requests, a nil body means the request has no
	// body, such as a GET request. The HTTP Client's Transport
	// is responsible for calling the Close method.
	//
	// For server requests, the Request Body is always non-nil
	// but will return EOF immediately when no body is present.
	// The Server will close the request body. The ServeHTTP
	// Handler does not need to.
	Body io.ReadCloser

	// GetBody defines an optional func to return a new copy of
	// Body. It is used for client requests when a redirect requires
	// reading the body more than once. Use of GetBody still
	// requires setting Body.
	//
	// For server requests, it is unused.
	GetBody func() (io.ReadCloser, error)

	// ContentLength records the length of the associated content.
	// The value -1 indicates that the length is unknown.
	// Values >= 0 indicate that the given number of bytes may
	// be read from Body.
	//
	// For client requests, a value of 0 with a non-nil Body is
	// also treated as unknown.
	ContentLength int64

	// TransferEncoding lists the transfer encodings from outermost to
	// innermost. An empty list denotes the "identity" encoding.
	// TransferEncoding can usually be ignored; chunked encoding is
	// automatically added and removed as necessary when sending and
	// receiving requests.
	TransferEncoding []string

	// Close indicates whether to close the connection after
	// replying to this request (for servers) or after sending this
	// request and reading its response (for clients).
	//
	// For server requests, the HTTP server handles this automatically
	// and this field is not needed by Handlers.
	//
	// For client requests, setting this field prevents re-use of
	// TCP connections between requests to the same hosts, as if
	// Transport.DisableKeepAlives were set.
	Close bool
```

```
// For server requests, Host specifies the host on which the
// URL is sought. For HTTP/1 (per RFC 7230, section 5.4), this
// is either the value of the "Host" header or the host name
// given in the URL itself. For HTTP/2, it is the value of the
// ":authority" pseudo-header field.
// It may be of the form "host:port". For international domain
// names, Host may be in Punycode or Unicode form. Use
// golang.org/x/net/idna to convert it to either format if
// needed.
// To prevent DNS rebinding attacks, server Handlers should
// validate that the Host header has a value for which the
// Handler considers itself authoritative. The included
// ServeMux supports patterns registered to particular host
// names and thus protects its registered Handlers.
//
// For client requests, Host optionally overrides the Host
// header to send. If empty, the Request.Write method uses
// the value of URL.Host. Host may contain an international
// domain name.
Host string

// Form contains the parsed form data, including both the URL
// field's query parameters and the PATCH, POST, or PUT form data.
// This field is only available after ParseForm is called.
// The HTTP client ignores Form and uses Body instead.
Form url.Values

// PostForm contains the parsed form data from PATCH, POST
// or PUT body parameters.
//
// This field is only available after ParseForm is called.
// The HTTP client ignores PostForm and uses Body instead.
PostForm url.Values

// MultipartForm is the parsed multipart form, including file uploads.
// This field is only available after ParseMultipartForm is called.
// The HTTP client ignores MultipartForm and uses Body instead.
MultipartForm *multipart.Form

// Trailer specifies additional headers that are sent after the request
// body.
//
// For server requests, the Trailer map initially contains only the
// trailer keys, with nil values. (The client declares which trailers it
// will later send.)  While the handler is reading from Body, it must
// not reference Trailer. After reading from Body returns EOF, Trailer
// can be read again and will contain non-nil values, if they were sent
// by the client.
//
// For client requests, Trailer must be initialized to a map containing
// the trailer keys to later send. The values may be nil or their final
// values. The ContentLength must be 0 or -1, to send a chunked request.
// After the HTTP request is sent the map values can be updated while
// the request body is read. Once the body returns EOF, the caller must
```

```
    // not mutate Trailer.
    //
    // Few HTTP clients, servers, or proxies support HTTP trailers.
    Trailer Header

    // RemoteAddr allows HTTP servers and other software to record
    // the network address that sent the request, usually for
    // logging. This field is not filled in by ReadRequest and
    // has no defined format. The HTTP server in this package
    // sets RemoteAddr to an "IP:port" address before invoking a
    // handler.
    // This field is ignored by the HTTP client.
    RemoteAddr string

    // RequestURI is the unmodified request-target of the
    // Request-Line (RFC 7230, Section 3.1.1) as sent by the client
    // to a server. Usually the URL field should be used instead.
    // It is an error to set this field in an HTTP client request.
    RequestURI string

    // TLS allows HTTP servers and other software to record
    // information about the TLS connection on which the request
    // was received. This field is not filled in by ReadRequest.
    // The HTTP server in this package sets the field for
    // TLS-enabled connections before invoking a handler;
    // otherwise it leaves the field nil.
    // This field is ignored by the HTTP client.
    TLS *tls.ConnectionState

    // Cancel is an optional channel whose closure indicates that the client
    // request should be regarded as canceled. Not all implementations of
    // RoundTripper may support Cancel.
    //
    // For server requests, this field is not applicable.
    //
    // Deprecated: Set the Request's context with NewRequestWithContext
    // instead. If a Request's Cancel field and context are both
    // set, it is undefined whether Cancel is respected.
    Cancel <-chan struct{}

    // Response is the redirect response which caused this request
    // to be created. This field is only populated during client
    // redirects.
    Response *Response
    // contains filtered or unexported fields
}
```

A Request represents an HTTP request received by a server or to be sent by a client.

The field semantics differ slightly between client and server usage. In addition to the notes on the fields below, see the documentation for Request.Write and RoundTripper.

func NewRequest

```
func NewRequest(method, url string, body io.Reader) (*Request, error)
```

NewRequest wraps NewRequestWithContext using the background context.

func NewRequestWithContext

```
func NewRequestWithContext(ctx context.Context, method, url string, body io.Reader) (
```

NewRequestWithContext returns a new Request given a method, URL, and optional body.

If the provided body is also an io.Closer, the returned Request.Body is set to body and will be closed by the Client methods Do, Post, and PostForm, and Transport.RoundTrip.

NewRequestWithContext returns a Request suitable for use with Client.Do or Transport.RoundTrip. To create a request for use with testing a Server Handler, either use the NewRequest function in the net/http/httptest package, use ReadRequest, or manually update the Request fields. For an outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body. See the Request type's documentation for the difference between inbound and outbound request fields.

If body is of type *bytes.Buffer, *bytes.Reader, or *strings.Reader, the returned request's ContentLength is set to its exact value (instead of -1), GetBody is populated (so 307 and 308 redirects can replay the body), and Body is set to NoBody if the ContentLength is 0.

func ReadRequest

```
func ReadRequest(b *bufio.Reader) (*Request, error)
```

ReadRequest reads and parses an incoming request from b.

ReadRequest is a low-level function and should only be used for specialized applications; most code should use the Server to read requests and handle them via the Handler interface. ReadRequest only supports HTTP/1.x requests. For HTTP/2, use golang.org/x/net/http2.

func (*Request) AddCookie

```
func (r *Request) AddCookie(c *Cookie)
```

AddCookie adds a cookie to the request. Per RFC 6265 section 5.4, AddCookie does not attach more than one Cookie header field. That means all cookies, if any, are written into the same line, separated by semicolon. AddCookie only sanitizes c's name and value, and does not sanitize a Cookie header already present in the request.

func (*Request) BasicAuth

```
func (r *Request) BasicAuth() (username, password string, ok bool)
```

BasicAuth returns the username and password provided in the request's Authorization header, if the request uses HTTP Basic Authentication. See RFC 2617, Section 2.

func (*Request) Clone

```
func (r *Request) Clone(ctx context.Context) *Request
```

Clone returns a deep copy of r with its context changed to ctx. The provided ctx must be non-nil.

For an outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body.

func (*Request) Context

```
func (r *Request) Context() context.Context
```

Context returns the request's context. To change the context, use WithContext.

The returned context is always non-nil; it defaults to the background context.

For outgoing client requests, the context controls cancellation.

For incoming server requests, the context is canceled when the client's connection closes, the request is canceled (with HTTP/2), or when the ServeHTTP method returns.

func (*Request) Cookie

```
func (r *Request) Cookie(name string) (*Cookie, error)
```

Cookie returns the named cookie provided in the request or ErrNoCookie if not found. If multiple cookies match the given name, only one cookie will be returned.

func (*Request) Cookies

```
func (r *Request) Cookies() []*Cookie
```

Cookies parses and returns the HTTP cookies sent with the request.

func (*Request) FormFile

```
func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

FormFile returns the first file for the provided form key. FormFile calls ParseMultipartForm and ParseForm if necessary.

func (*Request) FormValue

```
func (r *Request) FormValue(key string) string
```

FormValue returns the first value for the named component of the query. POST and PUT body parameters take precedence over URL query string values. FormValue calls ParseMultipartForm and ParseForm if necessary and ignores any errors returned by these functions. If key is not present, FormValue returns the empty string. To access multiple values of the same key, call ParseForm and then inspect Request.Form directly.

func (*Request) MultipartReader

```
func (r *Request) MultipartReader() (*multipart.Reader, error)
```

MultipartReader returns a MIME multipart reader if this is a multipart/form-data or a multipart/mixed POST request, else returns nil and an error. Use this function instead of ParseMultipartForm to process the request body as a stream.

func (*Request) ParseForm

```
func (r *Request) ParseForm() error
```

ParseForm populates r.Form and r.PostForm.

For all requests, ParseForm parses the raw query from the URL and updates r.Form.

For POST, PUT, and PATCH requests, it also reads the request body, parses it as a form and puts the results into both r.PostForm and r.Form. Request body parameters take precedence over URL query string values in r.Form.

If the request Body's size has not already been limited by MaxBytesReader, the size is capped at 10MB.

For other HTTP methods, or when the Content-Type is not application/x-www-form-urlencoded, the request Body is not read, and r.PostForm is initialized to a non-nil, empty value.

ParseMultipartForm calls ParseForm automatically. ParseForm is idempotent.

func (*Request) ParseMultipartForm

```
func (r *Request) ParseMultipartForm(maxMemory int64) error
```

ParseMultipartForm parses a request body as multipart/form-data. The whole request body is parsed and up to a total of maxMemory bytes of its file parts are stored in memory, with the remainder stored on disk in temporary files. ParseMultipartForm calls ParseForm if necessary. After one call to ParseMultipartForm, subsequent calls have no effect.

func (*Request) PostFormValue

```
func (r *Request) PostFormValue(key string) string
```

PostFormValue returns the first value for the named component of the POST, PATCH, or PUT request body. URL query parameters are ignored. PostFormValue calls ParseMultipartForm and ParseForm if necessary and ignores any errors returned by these functions. If key is not present, PostFormValue returns the empty string.

func (*Request) ProtoAtLeast

```
func (r *Request) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast reports whether the HTTP protocol used in the request is at least major.minor.

func (*Request) Referer

```
func (r *Request) Referer() string
```

Referer returns the referring URL, if sent in the request.

Referer is misspelled as in the request itself, a mistake from the earliest days of HTTP. This value can also be fetched from the Header map as Header["Referer"]; the benefit of making it available as a method is that the compiler can diagnose programs that use the alternate (correct English) spelling req.Referrer() but cannot diagnose programs that use Header["Referrer"].

func (*Request) SetBasicAuth

```
func (r *Request) SetBasicAuth(username, password string)
```

SetBasicAuth sets the request's Authorization header to use HTTP Basic Authentication with the provided username and password.

With HTTP Basic Authentication the provided username and password are not encrypted.

Some protocols may impose additional requirements on pre-escaping the username and password. For instance, when used with OAuth2, both arguments must be URL encoded first with url.QueryEscape.

func (*Request) UserAgent

```
func (r *Request) UserAgent() string
```

UserAgent returns the client's User-Agent, if sent in the request.

func (*Request) WithContext

```
func (r *Request) WithContext(ctx context.Context) *Request
```

WithContext returns a shallow copy of r with its context changed to ctx. The provided ctx must be non-nil.

For outgoing client request, the context controls the entire lifetime of a request and its response: obtaining a connection, sending the request, and reading the response headers and body.

To create a new request with a context, use NewRequestWithContext. To change the context of a request, such as an incoming request you want to modify before sending back out, use Request.Clone. Between those two uses, it's rare to need WithContext.

func (*Request) Write

```
func (r *Request) Write(w io.Writer) error
```

Write writes an HTTP/1.1 request, which is the header and body, in wire format. This method consults the following fields of the request:

```
Host
URL
Method (defaults to "GET")
Header
ContentLength
TransferEncoding
Body
```

If Body is present, Content-Length is <= 0 and TransferEncoding hasn't been set to "identity", Write adds "Transfer-Encoding: chunked" to the header. Body is closed after it is sent.

func (*Request) WriteProxy

```
func (r *Request) WriteProxy(w io.Writer) error
```

WriteProxy is like Write but writes the request in the form expected by an HTTP proxy. In particular, WriteProxy writes the initial Request-URI line of the request with an absolute URI, per section 5.3 of RFC 7230, including the scheme and host. In either case, WriteProxy also writes a Host header, using either r.Host or r.URL.Host.

type Response

```
type Response struct {
    Status     string // e.g. "200 OK"
    StatusCode int    // e.g. 200
    Proto      string // e.g. "HTTP/1.0"
    ProtoMajor int    // e.g. 1
    ProtoMinor int    // e.g. 0
```

```
// Header maps header keys to values. If the response had multiple
// headers with the same key, they may be concatenated, with comma
// delimiters.  (RFC 7230, section 3.2.2 requires that multiple headers
// be semantically equivalent to a comma-delimited sequence.) When
// Header values are duplicated by other fields in this struct (e.g.,
// ContentLength, TransferEncoding, Trailer), the field values are
// authoritative.
//
// Keys in the map are canonicalized (see CanonicalHeaderKey).
Header Header

// Body represents the response body.
//
// The response body is streamed on demand as the Body field
// is read. If the network connection fails or the server
// terminates the response, Body.Read calls return an error.
//
// The http Client and Transport guarantee that Body is always
// non-nil, even on responses without a body or responses with
// a zero-length body. It is the caller's responsibility to
// close Body. The default HTTP client's Transport may not
// reuse HTTP/1.x "keep-alive" TCP connections if the Body is
// not read to completion and closed.
//
// The Body is automatically dechunked if the server replied
// with a "chunked" Transfer-Encoding.
//
// As of Go 1.12, the Body will also implement io.Writer
// on a successful "101 Switching Protocols" response,
// as used by WebSockets and HTTP/2's "h2c" mode.
Body io.ReadCloser

// ContentLength records the length of the associated content. The
// value -1 indicates that the length is unknown. Unless Request.Method
// is "HEAD", values >= 0 indicate that the given number of bytes may
// be read from Body.
ContentLength int64

// Contains transfer encodings from outer-most to inner-most. Value is
// nil, means that "identity" encoding is used.
TransferEncoding []string

// Close records whether the header directed that the connection be
// closed after reading Body. The value is advice for clients: neither
// ReadResponse nor Response.Write ever closes a connection.
Close bool

// Uncompressed reports whether the response was sent compressed but
// was decompressed by the http package. When true, reading from
// Body yields the uncompressed content instead of the compressed
// content actually set from the server, ContentLength is set to -1,
// and the "Content-Length" and "Content-Encoding" fields are deleted
// from the responseHeader. To get the original response from
```

```
    // the server, set Transport.DisableCompression to true.
    Uncompressed bool

    // Trailer maps trailer keys to values in the same
    // format as Header.
    //
    // The Trailer initially contains only nil values, one for
    // each key specified in the server's "Trailer" header
    // value. Those values are not added to Header.
    //
    // Trailer must not be accessed concurrently with Read calls
    // on the Body.
    //
    // After Body.Read has returned io.EOF, Trailer will contain
    // any trailer values sent by the server.
    Trailer Header

    // Request is the request that was sent to obtain this Response.
    // Request's Body is nil (having already been consumed).
    // This is only populated for Client requests.
    Request *Request

    // TLS contains information about the TLS connection on which the
    // response was received. It is nil for unencrypted responses.
    // The pointer is shared between responses and should not be
    // modified.
    TLS *tls.ConnectionState
}
```

Response represents the response from an HTTP request.

The Client and Transport return Responses from servers once the response headers have been received. The response body is streamed on demand as the Body field is read.

func Get

```
func Get(url string) (resp *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect, up to a maximum of 10 redirects:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)
```

An error is returned if there were too many redirects or if there was an HTTP protocol error. A non-2xx response doesn't cause an error. Any returned error will be of type *url.Error. The url.Error value's Timeout method will report true if request timed out or was canceled.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

Get is a wrapper around DefaultClient.Get.

To make a request with custom headers, use NewRequest and DefaultClient.Do.

func Head

```
func Head(url string) (resp *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the following redirect codes, Head follows the redirect, up to a maximum of 10 redirects:

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
308 (Permanent Redirect)
```

Head is a wrapper around DefaultClient.Head

func Post

```
func Post(url, contentType string, body io.Reader) (resp *Response, err error)
```

Post issues a POST to the specified URL.

Caller should close resp.Body when done reading from it.

If the provided body is an io.Closer, it is closed after the request.

Post is a wrapper around DefaultClient.Post.

To set custom headers, use NewRequest and DefaultClient.Do.

See the Client.Do method documentation for details on how redirects are handled.

func PostForm

```
func PostForm(url string, data url.Values) (resp *Response, err error)
```

PostForm issues a POST to the specified URL, with data's keys and values URL-encoded as the request body.

The Content-Type header is set to application/x-www-form-urlencoded. To set other headers, use NewRequest and DefaultClient.Do.

When err is nil, resp always contains a non-nil resp.Body. Caller should close resp.Body when done reading from it.

PostForm is a wrapper around DefaultClient.PostForm.

See the Client.Do method documentation for details on how redirects are handled.

func ReadResponse

```
func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
```

ReadResponse reads and returns an HTTP response from r. The req parameter optionally specifies the Request that corresponds to this Response. If nil, a GET request is assumed. Clients must call resp.Body.Close when finished reading resp.Body. After that call, clients can inspect resp.Trailer to find key/value pairs included in the response trailer.

func (*Response) Cookies

```
func (r *Response) Cookies() []*Cookie
```

Cookies parses and returns the cookies set in the Set-Cookie headers.

func (*Response) Location

```
func (r *Response) Location() (*url.URL, error)
```

Location returns the URL of the response's "Location" header, if present. Relative redirects are resolved relative to the Response's Request. ErrNoLocation is returned if no Location header is present.

func (*Response) ProtoAtLeast

```
func (r *Response) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast reports whether the HTTP protocol used in the response is at least major.minor.

func (*Response) Write

```
func (r *Response) Write(w io.Writer) error
```

Write writes r to w in the HTTP/1.x server response format, including the status line, headers, body, and optional trailer.

This method consults the following fields of the response r:

```
StatusCode
ProtoMajor
ProtoMinor
```

```
Request.Method
TransferEncoding
Trailer
Body
ContentLength
Header, values for non-canonical keys will have unpredictable behavior
```

The Response Body is closed after it is sent.

type ResponseWriter

```
type ResponseWriter interface {
    // Header returns the header map that will be sent by
    // WriteHeader. The Header map also is the mechanism with which
    // Handlers can set HTTP trailers.
    //
    // Changing the header map after a call to WriteHeader (or
    // Write) has no effect unless the modified headers are
    // trailers.
    //
    // There are two ways to set Trailers. The preferred way is to
    // predeclare in the headers which trailers you will later
    // send by setting the "Trailer" header to the names of the
    // trailer keys which will come later. In this case, those
    // keys of the Header map are treated as if they were
    // trailers. See the example. The second way, for trailer
    // keys not known to the Handler until after the first Write,
    // is to prefix the Header map keys with the TrailerPrefix
    // constant value. See TrailerPrefix.
    //
    // To suppress automatic response headers (such as "Date"), set
    // their value to nil.
    Header() Header

    // Write writes the data to the connection as part of an HTTP reply.
    //
    // If WriteHeader has not yet been called, Write calls
    // WriteHeader(http.StatusOK) before writing the data. If the Header
    // does not contain a Content-Type line, Write adds a Content-Type set
    // to the result of passing the initial 512 bytes of written data to
    // DetectContentType. Additionally, if the total size of all written
    // data is under a few KB and there are no Flush calls, the
    // Content-Length header is added automatically.
    //
    // Depending on the HTTP protocol version and the client, calling
    // Write or WriteHeader may prevent future reads on the
    // Request.Body. For HTTP/1.x requests, handlers should read any
    // needed request body data before writing the response. Once the
    // headers have been flushed (due to either an explicit Flusher.Flush
    // call or writing enough data to trigger a flush), the request body
    // may be unavailable. For HTTP/2 requests, the Go HTTP server permits
    // handlers to continue to read the request body while concurrently
    // writing the response. However, such behavior may not be supported
```

```
    // by all HTTP/2 clients. Handlers should read before writing if
    // possible to maximize compatibility.
    Write([]byte) (int, error)

    // WriteHeader sends an HTTP response header with the provided
    // status code.
    //
    // If WriteHeader is not called explicitly, the first call to Write
    // will trigger an implicit WriteHeader(http.StatusOK).
    // Thus explicit calls to WriteHeader are mainly used to
    // send error codes.
    //
    // The provided code must be a valid HTTP 1xx-5xx status code.
    // Only one header may be written. Go does not currently
    // support sending user-defined 1xx informational headers,
    // with the exception of 100-continue response header that the
    // Server sends automatically when the Request.Body is read.
    WriteHeader(statusCode int)
}
```

A ResponseWriter interface is used by an HTTP handler to construct an HTTP response.

A ResponseWriter may not be used after the Handler.ServeHTTP method has returned.

type RoundTripper

```
type RoundTripper interface {
    // RoundTrip executes a single HTTP transaction, returning
    // a Response for the provided Request.
    //
    // RoundTrip should not attempt to interpret the response. In
    // particular, RoundTrip must return err == nil if it obtained
    // a response, regardless of the response's HTTP status code.
    // A non-nil err should be reserved for failure to obtain a
    // response. Similarly, RoundTrip should not attempt to
    // handle higher-level protocol details such as redirects,
    // authentication, or cookies.
    //
    // RoundTrip should not modify the request, except for
    // consuming and closing the Request's Body. RoundTrip may
    // read fields of the request in a separate goroutine. Callers
    // should not mutate or reuse the request until the Response's
    // Body has been closed.
    //
    // RoundTrip must always close the body, including on errors,
    // but depending on the implementation may do so in a separate
    // goroutine even after RoundTrip returns. This means that
    // callers wanting to reuse the body for subsequent requests
    // must arrange to wait for the Close call before doing so.
    //
    // The Request's URL and Header fields must be initialized.
    RoundTrip(*Request) (*Response, error)
}
```

RoundTripper is an interface representing the ability to execute a single HTTP transaction, obtaining the Response for a given Request.

A RoundTripper must be safe for concurrent use by multiple goroutines.

```
var DefaultTransport RoundTripper = &Transport{
    Proxy: ProxyFromEnvironment,
    DialContext: (&net.Dialer{
        Timeout:   30 * time.Second,
        KeepAlive: 30 * time.Second,
        DualStack: true,
    }).DialContext,
    ForceAttemptHTTP2:     true,
    MaxIdleConns:          100,
    IdleConnTimeout:       90 * time.Second,
    TLSHandshakeTimeout:   10 * time.Second,
    ExpectContinueTimeout: 1 * time.Second,
}
```

DefaultTransport is the default implementation of Transport and is used by DefaultClient. It establishes network connections as needed and caches them for reuse by subsequent calls. It uses HTTP proxies as directed by the $HTTP_PROXY and $NO_PROXY (or $http_proxy and $no_proxy) environment variables.

func NewFileTransport

```
func NewFileTransport(fs FileSystem) RoundTripper
```

NewFileTransport returns a new RoundTripper, serving the provided FileSystem. The returned RoundTripper ignores the URL host in its incoming requests, as well as most other properties of the request.

The typical use case for NewFileTransport is to register the "file" protocol with a Transport, as in:

```
t := &http.Transport{}
t.RegisterProtocol("file", http.NewFileTransport(http.Dir("/")))
c := &http.Client{Transport: t}
res, err := c.Get("file:///etc/passwd")
...
```

type SameSite

```
type SameSite int
```

SameSite allows a server to define a cookie attribute making it impossible for the browser to send this cookie along with cross-site requests. The main goal is to mitigate the risk of cross-origin information leakage, and provide some protection against cross-site request forgery attacks.

See https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00 for details.

```
const (
    SameSiteDefaultMode SameSite = iota + 1
    SameSiteLaxMode
    SameSiteStrictMode
    SameSiteNoneMode
)
```

type ServeMux

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

ServeMux is an HTTP request multiplexer. It matches the URL of each incoming request against a list of registered patterns and calls the handler for the pattern that most closely matches the URL.

Patterns name fixed, rooted paths, like "/favicon.ico", or rooted subtrees, like "/images/" (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "/images/" and "/images/thumbnails/", the latter handler will be called for paths beginning "/images/thumbnails/" and the former will receive requests for any other paths in the "/images/" subtree.

Note that since a pattern ending in a slash names a rooted subtree, the pattern "/" matches all paths not matched by other registered patterns, not just the URL with Path == "/".

If a subtree has been registered and a request is received naming the subtree root without its trailing slash, ServeMux redirects that request to the subtree root (adding the trailing slash). This behavior can be overridden with a separate registration for the path without the trailing slash. For example, registering "/images/" causes ServeMux to redirect a request for "/images" to "/images/", unless "/images" has been registered separately.

Patterns may optionally begin with a host name, restricting matches to URLs on that host only. Host-specific patterns take precedence over general patterns, so that a handler might register for the two patterns "/codesearch" and "codesearch.google.com/" without also taking over requests for "http://www.google.com/".

ServeMux also takes care of sanitizing the URL request path and the Host header, stripping the port number and redirecting any request containing . or .. elements or repeated slashes to an equivalent, cleaner URL.

func NewServeMux

```
func NewServeMux() *ServeMux
```

NewServeMux allocates and returns a new ServeMux.

func (*ServeMux) Handle

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern. If a handler already exists for pattern, Handle panics.

func (*ServeMux) HandleFunc

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request
```

HandleFunc registers the handler function for the given pattern.

func (*ServeMux) Handler

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
```

Handler returns the handler to use for the given request, consulting r.Method, r.Host, and r.URL.Path. It always returns a non-nil handler. If the path is not in its canonical form, the handler will be an internally-generated handler that redirects to the canonical path. If the host contains a port, it is ignored when matching handlers.

The path and host are used unchanged for CONNECT requests.

Handler also returns the registered pattern that matches the request or, in the case of internally-generated redirects, the pattern that will match after following the redirect.

If there is no registered handler that applies to the request, Handler returns a ``page not found" handler and an empty pattern.

func (*ServeMux) ServeHTTP

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP dispatches the request to the handler whose pattern most closely matches the request URL.

type Server

```
type Server struct {
    // Addr optionally specifies the TCP address for the server to listen on,
    // in the form "host:port". If empty, ":http" (port 80) is used.
    // The service names are defined in RFC 6335 and assigned by IANA.
    // See net.Dial for details of the address format.
    Addr string

    Handler Handler // handler to invoke, http.DefaultServeMux if nil
```

```
// TLSConfig optionally provides a TLS configuration for use
// by ServeTLS and ListenAndServeTLS. Note that this value is
// cloned by ServeTLS and ListenAndServeTLS, so it's not
// possible to modify the configuration with methods like
// tls.Config.SetSessionTicketKeys. To use
// SetSessionTicketKeys, use Server.Serve with a TLS Listener
// instead.
TLSConfig *tls.Config

// ReadTimeout is the maximum duration for reading the entire
// request, including the body.
//
// Because ReadTimeout does not let Handlers make per-request
// decisions on each request body's acceptable deadline or
// upload rate, most users will prefer to use
// ReadHeaderTimeout. It is valid to use them both.
ReadTimeout time.Duration

// ReadHeaderTimeout is the amount of time allowed to read
// request headers. The connection's read deadline is reset
// after reading the headers and the Handler can decide what
// is considered too slow for the body. If ReadHeaderTimeout
// is zero, the value of ReadTimeout is used. If both are
// zero, there is no timeout.
ReadHeaderTimeout time.Duration

// WriteTimeout is the maximum duration before timing out
// writes of the response. It is reset whenever a new
// request's header is read. Like ReadTimeout, it does not
// let Handlers make decisions on a per-request basis.
WriteTimeout time.Duration

// IdleTimeout is the maximum amount of time to wait for the
// next request when keep-alives are enabled. If IdleTimeout
// is zero, the value of ReadTimeout is used. If both are
// zero, there is no timeout.
IdleTimeout time.Duration

// MaxHeaderBytes controls the maximum number of bytes the
// server will read parsing the request header's keys and
// values, including the request line. It does not limit the
// size of the request body.
// If zero, DefaultMaxHeaderBytes is used.
MaxHeaderBytes int

// TLSNextProto optionally specifies a function to take over
// ownership of the provided TLS connection when an ALPN
// protocol upgrade has occurred. The map key is the protocol
// name negotiated. The Handler argument should be used to
// handle HTTP requests and will initialize the Request's TLS
// and RemoteAddr if not already set. The connection is
// automatically closed when the function returns.
// If TLSNextProto is not nil, HTTP/2 support is not enabled
// automatically.
```

```
    TLSNextProto map[string]func(*Server, *tls.Conn, Handler)

    // ConnState specifies an optional callback function that is
    // called when a client connection changes state. See the
    // ConnState type and associated constants for details.
    ConnState func(net.Conn, ConnState)

    // ErrorLog specifies an optional logger for errors accepting
    // connections, unexpected behavior from handlers, and
    // underlying FileSystem errors.
    // If nil, logging is done via the log package's standard logger.
    ErrorLog *log.Logger

    // BaseContext optionally specifies a function that returns
    // the base context for incoming requests on this server.
    // The provided Listener is the specific Listener that's
    // about to start accepting requests.
    // If BaseContext is nil, the default is context.Background().
    // If non-nil, it must return a non-nil context.
    BaseContext func(net.Listener) context.Context

    // ConnContext optionally specifies a function that modifies
    // the context used for a new connection c. The provided ctx
    // is derived from the base context and has a ServerContextKey
    // value.
    ConnContext func(ctx context.Context, c net.Conn) context.Context
    // contains filtered or unexported fields
}
```

A Server defines parameters for running an HTTP server. The zero value for Server is a valid configuration.

func (*Server) Close

```
func (srv *Server) Close() error
```

Close immediately closes all active net.Listeners and any connections in state StateNew, StateActive, or StateIdle. For a graceful shutdown, use Shutdown.

Close does not attempt to close (and does not even know about) any hijacked connections, such as WebSockets.

Close returns any error returned from closing the Server's underlying Listener(s).

func (*Server) ListenAndServe

```
func (srv *Server) ListenAndServe() error
```

ListenAndServe listens on the TCP network address srv.Addr and then calls Serve to handle requests on incoming connections. Accepted connections are configured to enable TCP keep-alives.

If srv.Addr is blank, ":http" is used.

ListenAndServe always returns a non-nil error. After Shutdown or Close, the returned error is ErrServerClosed.

func (*Server) ListenAndServeTLS

```
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
```

ListenAndServeTLS listens on the TCP network address srv.Addr and then calls ServeTLS to handle requests on incoming TLS connections. Accepted connections are configured to enable TCP keep-alives.

Filenames containing a certificate and matching private key for the server must be provided if neither the Server's TLSConfig.Certificates nor TLSConfig.GetCertificate are populated. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

If srv.Addr is blank, ":https" is used.

ListenAndServeTLS always returns a non-nil error. After Shutdown or Close, the returned error is ErrServerClosed.

func (*Server) RegisterOnShutdown

```
func (srv *Server) RegisterOnShutdown(f func())
```

RegisterOnShutdown registers a function to call on Shutdown. This can be used to gracefully shutdown connections that have undergone ALPN protocol upgrade or that have been hijacked. This function should start protocol-specific graceful shutdown, but should not wait for shutdown to complete.

func (*Server) Serve

```
func (srv *Server) Serve(l net.Listener) error
```

Serve accepts incoming connections on the Listener l, creating a new service goroutine for each. The service goroutines read requests and then call srv.Handler to reply to them.

HTTP/2 support is only enabled if the Listener returns *tls.Conn connections and they were configured with "h2" in the TLS Config.NextProtos.

Serve always returns a non-nil error and closes l. After Shutdown or Close, the returned error is ErrServerClosed.

func (*Server) ServeTLS

```
func (srv *Server) ServeTLS(l net.Listener, certFile, keyFile string) error
```

ServeTLS accepts incoming connections on the Listener l, creating a new service goroutine for each. The service goroutines perform TLS setup and then read requests, calling srv.Handler to reply to them.

Files containing a certificate and matching private key for the server must be provided if neither the Server's TLSConfig.Certificates nor TLSConfig.GetCertificate are populated. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate, any intermediates, and the CA's certificate.

ServeTLS always returns a non-nil error. After Shutdown or Close, the returned error is ErrServerClosed.

func (*Server) SetKeepAlivesEnabled

```
func (srv *Server) SetKeepAlivesEnabled(v bool)
```

SetKeepAlivesEnabled controls whether HTTP keep-alives are enabled. By default, keep-alives are always enabled. Only very resource-constrained environments or servers in the process of shutting down should disable them.

func (*Server) Shutdown

```
func (srv *Server) Shutdown(ctx context.Context) error
```

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down. If the provided context expires before the shutdown is complete, Shutdown returns the context's error, otherwise it returns any error returned from closing the Server's underlying Listener(s).

When Shutdown is called, Serve, ListenAndServe, and ListenAndServeTLS immediately return ErrServerClosed. Make sure the program doesn't exit and waits instead for Shutdown to return.

Shutdown does not attempt to close nor wait for hijacked connections such as WebSockets. The caller of Shutdown should separately notify such long-lived connections of shutdown and wait for them to close, if desired. See RegisterOnShutdown for a way to register shutdown notification functions.

Once Shutdown has been called on a server, it may not be reused; future calls to methods such as Serve will return ErrServerClosed.

type Transport

```
type Transport struct {

    // Proxy specifies a function to return a proxy for a given
```

```
    // Request. If the function returns a non-nil error, the
    // request is aborted with the provided error.
    //
    // The proxy type is determined by the URL scheme. "http",
    // "https", and "socks5" are supported. If the scheme is empty,
    // "http" is assumed.
    //
    // If Proxy is nil or returns a nil *URL, no proxy is used.
    Proxy func(*Request) (*url.URL, error)

    // DialContext specifies the dial function for creating unencrypted TCP connectio
    // If DialContext is nil (and the deprecated Dial below is also nil),
    // then the transport dials using package net.
    //
    // DialContext runs concurrently with calls to RoundTrip.
    // A RoundTrip call that initiates a dial may end up using
    // a connection dialed previously when the earlier connection
    // becomes idle before the later DialContext completes.
    DialContext func(ctx context.Context, network, addr string) (net.Conn, error)

    // Dial specifies the dial function for creating unencrypted TCP connections.
    //
    // Dial runs concurrently with calls to RoundTrip.
    // A RoundTrip call that initiates a dial may end up using
    // a connection dialed previously when the earlier connection
    // becomes idle before the later Dial completes.
    //
    // Deprecated: Use DialContext instead, which allows the transport
    // to cancel dials as soon as they are no longer needed.
    // If both are set, DialContext takes priority.
    Dial func(network, addr string) (net.Conn, error)

    // DialTLSContext specifies an optional dial function for creating
    // TLS connections for non-proxied HTTPS requests.
    //
    // If DialTLSContext is nil (and the deprecated DialTLS below is also nil),
    // DialContext and TLSClientConfig are used.
    //
    // If DialTLSContext is set, the Dial and DialContext hooks are not used for HTTP
    // requests and the TLSClientConfig and TLSHandshakeTimeout
    // are ignored. The returned net.Conn is assumed to already be
    // past the TLS handshake.
    DialTLSContext func(ctx context.Context, network, addr string) (net.Conn, error)

    // DialTLS specifies an optional dial function for creating
    // TLS connections for non-proxied HTTPS requests.
    //
    // Deprecated: Use DialTLSContext instead, which allows the transport
    // to cancel dials as soon as they are no longer needed.
    // If both are set, DialTLSContext takes priority.
    DialTLS func(network, addr string) (net.Conn, error)

    // TLSClientConfig specifies the TLS configuration to use with
    // tls.Client.
```

```
    // If nil, the default configuration is used.
    // If non-nil, HTTP/2 support may not be enabled by default.
    TLSClientConfig *tls.Config

    // TLSHandshakeTimeout specifies the maximum amount of time waiting to
    // wait for a TLS handshake. Zero means no timeout.
    TLSHandshakeTimeout time.Duration

    // DisableKeepAlives, if true, disables HTTP keep-alives and
    // will only use the connection to the server for a single
    // HTTP request.
    //
    // This is unrelated to the similarly named TCP keep-alives.
    DisableKeepAlives bool

    // DisableCompression, if true, prevents the Transport from
    // requesting compression with an "Accept-Encoding: gzip"
    // request header when the Request contains no existing
    // Accept-Encoding value. If the Transport requests gzip on
    // its own and gets a gzipped response, it's transparently
    // decoded in the Response.Body. However, if the user
    // explicitly requested gzip it is not automatically
    // uncompressed.
    DisableCompression bool

    // MaxIdleConns controls the maximum number of idle (keep-alive)
    // connections across all hosts. Zero means no limit.
    MaxIdleConns int

    // MaxIdleConnsPerHost, if non-zero, controls the maximum idle
    // (keep-alive) connections to keep per-host. If zero,
    // DefaultMaxIdleConnsPerHost is used.
    MaxIdleConnsPerHost int

    // MaxConnsPerHost optionally limits the total number of
    // connections per host, including connections in the dialing,
    // active, and idle states. On limit violation, dials will block.
    //
    // Zero means no limit.
    MaxConnsPerHost int

    // IdleConnTimeout is the maximum amount of time an idle
    // (keep-alive) connection will remain idle before closing
    // itself.
    // Zero means no limit.
    IdleConnTimeout time.Duration

    // ResponseHeaderTimeout, if non-zero, specifies the amount of
    // time to wait for a server's response headers after fully
    // writing the request (including its body, if any). This
    // time does not include the time to read the response body.
    ResponseHeaderTimeout time.Duration

    // ExpectContinueTimeout, if non-zero, specifies the amount of
```

```
    // time to wait for a server's first response headers after fully
    // writing the request headers if the request has an
    // "Expect: 100-continue" header. Zero means no timeout and
    // causes the body to be sent immediately, without
    // waiting for the server to approve.
    // This time does not include the time to send the request header.
    ExpectContinueTimeout time.Duration

    // TLSNextProto specifies how the Transport switches to an
    // alternate protocol (such as HTTP/2) after a TLS ALPN
    // protocol negotiation. If Transport dials an TLS connection
    // with a non-empty protocol name and TLSNextProto contains a
    // map entry for that key (such as "h2"), then the func is
    // called with the request's authority (such as "example.com"
    // or "example.com:1234") and the TLS connection. The function
    // must return a RoundTripper that then handles the request.
    // If TLSNextProto is not nil, HTTP/2 support is not enabled
    // automatically.
    TLSNextProto map[string]func(authority string, c *tls.Conn) RoundTripper

    // ProxyConnectHeader optionally specifies headers to send to
    // proxies during CONNECT requests.
    ProxyConnectHeader Header

    // MaxResponseHeaderBytes specifies a limit on how many
    // response bytes are allowed in the server's response
    // header.
    //
    // Zero means to use a default limit.
    MaxResponseHeaderBytes int64

    // WriteBufferSize specifies the size of the write buffer used
    // when writing to the transport.
    // If zero, a default (currently 4KB) is used.
    WriteBufferSize int

    // ReadBufferSize specifies the size of the read buffer used
    // when reading from the transport.
    // If zero, a default (currently 4KB) is used.
    ReadBufferSize int

    // ForceAttemptHTTP2 controls whether HTTP/2 is enabled when a non-zero
    // Dial, DialTLS, or DialContext func or TLSClientConfig is provided.
    // By default, use of any those fields conservatively disables HTTP/2.
    // To use a custom dialer or TLS config and still attempt HTTP/2
    // upgrades, set this to true.
    ForceAttemptHTTP2 bool
    // contains filtered or unexported fields
}
```

Transport is an implementation of RoundTripper that supports HTTP, HTTPS, and HTTP proxies (for either HTTP or HTTPS with CONNECT).

By default, Transport caches connections for future re-use. This may leave many open connections when accessing many hosts. This behavior can be managed using Transport's CloseIdleConnections method and the MaxIdleConnsPerHost and DisableKeepAlives fields.

Transports should be reused instead of created as needed. Transports are safe for concurrent use by multiple goroutines.

A Transport is a low-level primitive for making HTTP and HTTPS requests. For high-level functionality, such as cookies and redirects, see Client.

Transport uses HTTP/1.1 for HTTP URLs and either HTTP/1.1 or HTTP/2 for HTTPS URLs, depending on whether the server supports HTTP/2, and how the Transport is configured. The DefaultTransport supports HTTP/2. To explicitly enable HTTP/2 on a transport, use golang.org/x/net/http2 and call ConfigureTransport. See the package docs for more about HTTP/2.

Responses with status codes in the 1xx range are either handled automatically (100 expect-continue) or ignored. The one exception is HTTP status code 101 (Switching Protocols), which is considered a terminal status and returned by RoundTrip. To see the ignored 1xx responses, use the httptrace trace package's ClientTrace.Got1xxResponse.

Transport only retries a request upon encountering a network error if the request is idempotent and either has no body or has its Request.GetBody defined. HTTP requests are considered idempotent if they have HTTP methods GET, HEAD, OPTIONS, or TRACE; or if their Header map contains an "Idempotency-Key" or "X-Idempotency-Key" entry. If the idempotency key value is a zero-length slice, the request is treated as idempotent but the header is not sent on the wire.

func (*Transport) CancelRequest

```
func (t *Transport) CancelRequest(req *Request)
```

CancelRequest cancels an in-flight request by closing its connection. CancelRequest should only be called after RoundTrip has returned.

Deprecated: Use Request.WithContext to create a request with a cancelable context instead. CancelRequest cannot cancel HTTP/2 requests.

func (*Transport) Clone

```
func (t *Transport) Clone() *Transport
```

Clone returns a deep copy of t's exported fields.

func (*Transport) CloseIdleConnections

```
func (t *Transport) CloseIdleConnections()
```

CloseIdleConnections closes any connections which were previously connected from previous requests but are now sitting idle in a "keep-alive" state. It does not interrupt any connections currently in use.

func (*Transport) RegisterProtocol

```
func (t *Transport) RegisterProtocol(scheme string, rt RoundTripper)
```

RegisterProtocol registers a new protocol with scheme. The Transport will pass requests using the given scheme to rt. It is rt's responsibility to simulate HTTP request semantics.

RegisterProtocol can be used by other packages to provide implementations of protocol schemes like "ftp" or "file".

If rt.RoundTrip returns ErrSkipAltProtocol, the Transport will handle the RoundTrip itself for that one request, as if the protocol were not registered.

func (*Transport) RoundTrip

```
func (t *Transport) RoundTrip(req *Request) (*Response, error)
```

RoundTrip implements the RoundTripper interface.

For higher-level HTTP client support (such as handling of cookies and redirects), see Get, Post, and the Client type.

Like the RoundTripper interface, the error types returned by RoundTrip are unspecified.