net

Package net go1.15.2 Latest

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

Overview

Package net provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets.

Although the package provides access to low-level networking primitives, most clients will need only the basic interface provided by the Dial, Listen, and Accept functions and the associated Conn and Listener interfaces. The crypto/tls package uses the same interfaces and similar Dial and Listen functions.

The Dial function connects to a server:

```
conn, err := net.Dial("tcp", "golang.org:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

The Listen function creates servers:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

Name Resolution

The method for resolving domain names, whether indirectly with functions like Dial or directly with functions like LookupHost and LookupAddr, varies by operating system.

On Unix systems, the resolver has two options for resolving names. It can use a pure Go resolver that sends DNS requests directly to the servers listed in /etc/resolv.conf, or it can use a cgo-based resolver

that calls C library routines such as getaddrinfo and getnameinfo.

By default the pure Go resolver is used, because a blocked DNS request consumes only a goroutine, while a blocked C call consumes an operating system thread. When cgo is available, the cgo-based resolver is used instead under a variety of conditions: on systems that do not let programs make direct DNS requests (OS X), when the LOCALDOMAIN environment variable is present (even if empty), when the RES_OPTIONS or HOSTALIASES environment variable is non-empty, when the ASR_CONFIG environment variable is non-empty (OpenBSD only), when /etc/resolv.conf or /etc/nsswitch.conf specify the use of features that the Go resolver does not implement, and when the name being looked up ends in .local or is an mDNS name.

The resolver decision can be overridden by setting the netdns value of the GODEBUG environment variable (see package runtime) to go or cgo, as in:

```
export GODEBUG=netdns=go    # force pure Go resolver
export GODEBUG=netdns=cgo   # force cgo resolver
```

The decision can also be forced while building the Go source tree by setting the netgo or netcgo build tag.

A numeric netdns setting, as in GODEBUG=netdns=1, causes the resolver to print debugging information about its decisions. To force a particular resolver while also printing debugging information, join the two settings by a plus sign, as in GODEBUG=netdns=go+1.

On Plan 9, the resolver always accesses /net/cs and /net/dns.

On Windows, the resolver always uses C library functions, such as GetAddrInfo and DnsQuery.

Constants

```
const (
    IPv4len = 4
    IPv6len = 16
)
```

IP address lengths (bytes).

Variables

```
var (
    IPv4bcast     = IPv4(255, 255, 255, 255) // limited broadcast
    IPv4allsys    = IPv4(224, 0, 0, 1)       // all systems
    IPv4allrouter = IPv4(224, 0, 0, 2)       // all routers
    IPv4zero      = IPv4(0, 0, 0, 0)         // all zeros
)
```

Well-known IPv4 addresses

```
var (
    IPv6zero                   = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    IPv6unspecified            = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    IPv6loopback               = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
    IPv6interfacelocalallnodes = IP{0xff, 0x01, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    IPv6linklocalallnodes      = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    IPv6linklocalallrouters    = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
)
```

Well-known IPv6 addresses

```
var DefaultResolver = &Resolver{}
```

DefaultResolver is the resolver used by the package-level Lookup functions and by Dialers without a specified Resolver.

```
var (
    ErrWriteToConnected = errors.New("use of WriteTo with pre-connected connection")
)
```

Various errors contained in OpError.

func JoinHostPort

```
func JoinHostPort(host, port string) string
```

JoinHostPort combines host and port into a network address of the form "host:port". If host contains a colon, as found in literal IPv6 addresses, then JoinHostPort returns "[host]:port".

See func Dial for a description of the host and port parameters.

func LookupAddr

```
func LookupAddr(addr string) (names []string, err error)
```

LookupAddr performs a reverse lookup for the given address, returning a list of names mapping to that address.

When using the host C library resolver, at most one result will be returned. To bypass the host resolver, use a custom Resolver.

func LookupCNAME

```
func LookupCNAME(host string) (cname string, err error)
```

LookupCNAME returns the canonical name for the given host. Callers that do not care about the canonical name can call LookupHost or LookupIP directly; both take care of resolving the canonical

name as part of the lookup.

A canonical name is the final name after following zero or more CNAME records. LookupCNAME does not return an error if host does not contain DNS "CNAME" records, as long as host resolves to address records.

func LookupHost

```
func LookupHost(host string) (addrs []string, err error)
```

LookupHost looks up the given host using the local resolver. It returns a slice of that host's addresses.

func LookupPort

```
func LookupPort(network, service string) (port int, err error)
```

LookupPort looks up the port for the given network and service.

func LookupTXT

```
func LookupTXT(name string) ([]string, error)
```

LookupTXT returns the DNS TXT records for the given domain name.

func ParseCIDR

```
func ParseCIDR(s string) (IP, *IPNet, error)
```

ParseCIDR parses s as a CIDR notation IP address and prefix length, like "192.0.2.0/24" or "2001:db8::/32", as defined in RFC 4632 and RFC 4291.

It returns the IP address and the network implied by the IP and prefix length. For example, ParseCIDR("192.0.2.1/24") returns the IP address 192.0.2.1 and the network 192.0.2.0/24.

func Pipe

```
func Pipe() (Conn, Conn)
```

Pipe creates a synchronous, in-memory, full duplex network connection; both ends implement the Conn interface. Reads on one end are matched with writes on the other, copying data directly between the two; there is no internal buffering.

func SplitHostPort

```
func SplitHostPort(hostport string) (host, port string, err error)
```

SplitHostPort splits a network address of the form "host:port", "host%zone:port", "[host]:port" or "[host%zone]:port" into host or host%zone and port.

A literal IPv6 address in hostport must be enclosed in square brackets, as in "[::1]:80", "[::1%lo0]:80".

See func Dial for a description of the hostport parameter, and host and port results.

type Addr

```
type Addr interface {
    Network() string // name of the network (for example, "tcp", "udp")
    String() string  // string form of address (for example, "192.0.2.1:25", "[2001:d
}
```

Addr represents a network end point address.

The two methods Network and String conventionally return strings that can be passed as the arguments to Dial, but the exact form and meaning of the strings is up to the implementation.

func InterfaceAddrs

```
func InterfaceAddrs() ([]Addr, error)
```

InterfaceAddrs returns a list of the system's unicast interface addresses.

The returned list does not identify the associated interface; use Interfaces and Interface.Addrs for more detail.

type AddrError

```
type AddrError struct {
    Err  string
    Addr string
}
```

func (*AddrError) Error

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

func (*AddrError) Temporary

```
func (e *AddrError) Temporary() bool
```

func (*AddrError) Timeout

```
func (e *AddrError) Timeout() bool
```

type Buffers

```
type Buffers [][]byte
```

Buffers contains zero or more runs of bytes to write.

On certain machines, for certain types of connections, this is optimized into an OS-specific batch write operation (such as "writev").

func (*Buffers) Read

```
func (v *Buffers) Read(p []byte) (n int, err error)
```

func (*Buffers) WriteTo

```
func (v *Buffers) WriteTo(w io.Writer) (n int64, err error)
```

type Conn

```
type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return an error after a fixed
    // time limit; see SetDeadline and SetReadDeadline.
    Read(b []byte) (n int, err error)

    // Write writes data to the connection.
    // Write can be made to time out and return an error after a fixed
    // time limit; see SetDeadline and SetWriteDeadline.
    Write(b []byte) (n int, err error)

    // Close closes the connection.
    // Any blocked Read or Write operations will be unblocked and return errors.
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // RemoteAddr returns the remote network address.
    RemoteAddr() Addr

    // SetDeadline sets the read and write deadlines associated
    // with the connection. It is equivalent to calling both
    // SetReadDeadline and SetWriteDeadline.
    //
    // A deadline is an absolute time after which I/O operations
    // fail instead of blocking. The deadline applies to all future
    // and pending I/O, not just the immediately following call to
    // Read or Write. After a deadline has been exceeded, the
    // connection can be refreshed by setting a deadline in the future.
    //
```

```
    // If the deadline is exceeded a call to Read or Write or to other
    // I/O methods will return an error that wraps os.ErrDeadlineExceeded.
    // This can be tested using errors.Is(err, os.ErrDeadlineExceeded).
    // The error's Timeout method will return true, but note that there
    // are other possible errors for which the Timeout method will
    // return true even if the deadline has not been exceeded.
    //
    // An idle timeout can be implemented by repeatedly extending
    // the deadline after successful Read or Write calls.
    //
    // A zero value for t means I/O operations will not time out.
    SetDeadline(t time.Time) error

    // SetReadDeadline sets the deadline for future Read calls
    // and any currently-blocked Read call.
    // A zero value for t means Read will not time out.
    SetReadDeadline(t time.Time) error

    // SetWriteDeadline sets the deadline for future Write calls
    // and any currently-blocked Write call.
    // Even if write times out, it may return n > 0, indicating that
    // some of the data was successfully written.
    // A zero value for t means Write will not time out.
    SetWriteDeadline(t time.Time) error
}
```

Conn is a generic stream-oriented network connection.

Multiple goroutines may invoke methods on a Conn simultaneously.

func Dial

```
func Dial(network, address string) (Conn, error)
```

Dial connects to the address on the named network.

Known networks are "tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only), "ip6" (IPv6-only), "unix", "unixgram" and "unixpacket".

For TCP and UDP networks, the address has the form "host:port". The host must be a literal IP address, or a host name that can be resolved to IP addresses. The port must be a literal port number or a service name. If the host is a literal IPv6 address it must be enclosed in square brackets, as in "[2001:db8::1]:80" or "[fe80::1%zone]:80". The zone specifies the scope of the literal IPv6 address as defined in RFC 4007. The functions JoinHostPort and SplitHostPort manipulate a pair of host and port in this form. When using TCP, and the host resolves to multiple IP addresses, Dial will try each IP address in order until one succeeds.

Examples:

```
Dial("tcp", "golang.org:http")
Dial("tcp", "192.0.2.1:http")
Dial("tcp", "198.51.100.1:80")
Dial("udp", "[2001:db8::1]:domain")
Dial("udp", "[fe80::1%lo0]:53")
Dial("tcp", ":80")
```

For IP networks, the network must be "ip", "ip4" or "ip6" followed by a colon and a literal protocol number or a protocol name, and the address has the form "host". The host must be a literal IP address or a literal IPv6 address with zone. It depends on each operating system how the operating system behaves with a non-well known protocol number such as "0" or "255".

Examples:

```
Dial("ip4:1", "192.0.2.1")
Dial("ip6:ipv6-icmp", "2001:db8::1")
Dial("ip6:58", "fe80::1%lo0")
```

For TCP, UDP and IP networks, if the host is empty or a literal unspecified IP address, as in ":80", "0.0.0.0:80" or "[::]:80" for TCP and UDP, "", "0.0.0.0" or "::" for IP, the local system is assumed.

For Unix networks, the address must be a file system path.

func DialTimeout

```
func DialTimeout(network, address string, timeout time.Duration) (Conn, error)
```

DialTimeout acts like Dial but takes a timeout.

The timeout includes name resolution, if required. When using TCP, and the host in the address parameter resolves to multiple IP addresses, the timeout is spread over each consecutive dial, such that each is given an appropriate fraction of the time to connect.

See func Dial for a description of the network and address parameters.

func FileConn

```
func FileConn(f *os.File) (c Conn, err error)
```

FileConn returns a copy of the network connection corresponding to the open file f. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

type DNSConfigError

```
type DNSConfigError struct {
    Err error
}
```

DNSConfigError represents an error reading the machine's DNS configuration. (No longer used; kept for compatibility.)

func (*DNSConfigError) Error

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

func (*DNSConfigError) Temporary

```
func (e *DNSConfigError) Temporary() bool
```

func (*DNSConfigError) Timeout

```
func (e *DNSConfigError) Timeout() bool
```

func (*DNSConfigError) Unwrap

```
func (e *DNSConfigError) Unwrap() error
```

type DNSError

```
type DNSError struct {
    Err         string // description of the error
    Name        string // name looked for
    Server      string // server used
    IsTimeout   bool   // if true, timed out; not all timeouts set this
    IsTemporary bool   // if true, error is temporary; not all errors set this
    IsNotFound  bool   // if true, host could not be found
}
```

DNSError represents a DNS lookup error.

func (*DNSError) Error

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

func (*DNSError) Temporary

```
func (e *DNSError) Temporary() bool
```

Temporary reports whether the DNS error is known to be temporary. This is not always known; a DNS lookup may fail due to a temporary error and return a DNSError for which Temporary returns false.

func (*DNSError) Timeout

```
func (e *DNSError) Timeout() bool
```

Timeout reports whether the DNS lookup is known to have timed out. This is not always known; a DNS lookup may fail due to a timeout and return a DNSError for which Timeout returns false.

type Dialer

```
type Dialer struct {
    // Timeout is the maximum amount of time a dial will wait for
    // a connect to complete. If Deadline is also set, it may fail
    // earlier.
    //
    // The default is no timeout.
    //
    // When using TCP and dialing a host name with multiple IP
    // addresses, the timeout may be divided between them.
    //
    // With or without a timeout, the operating system may impose
    // its own earlier timeout. For instance, TCP timeouts are
    // often around 3 minutes.
    Timeout time.Duration

    // Deadline is the absolute point in time after which dials
    // will fail. If Timeout is set, it may fail earlier.
    // Zero means no deadline, or dependent on the operating system
    // as with the Timeout option.
    Deadline time.Time

    // LocalAddr is the local address to use when dialing an
    // address. The address must be of a compatible type for the
    // network being dialed.
    // If nil, a local address is automatically chosen.
    LocalAddr Addr

    // DualStack previously enabled RFC 6555 Fast Fallback
    // support, also known as "Happy Eyeballs", in which IPv4 is
    // tried soon if IPv6 appears to be misconfigured and
    // hanging.
    //
    // Deprecated: Fast Fallback is enabled by default. To
    // disable, set FallbackDelay to a negative value.
    DualStack bool

    // FallbackDelay specifies the length of time to wait before
    // spawning a RFC 6555 Fast Fallback connection. That is, this
    // is the amount of time to wait for IPv6 to succeed before
    // assuming that IPv6 is misconfigured and falling back to
    // IPv4.
    //
    // If zero, a default delay of 300ms is used.
    // A negative value disables Fast Fallback support.
    FallbackDelay time.Duration
```

```
    // KeepAlive specifies the interval between keep-alive
    // probes for an active network connection.
    // If zero, keep-alive probes are sent with a default value
    // (currently 15 seconds), if supported by the protocol and operating
    // system. Network protocols or operating systems that do
    // not support keep-alives ignore this field.
    // If negative, keep-alive probes are disabled.
    KeepAlive time.Duration

    // Resolver optionally specifies an alternate resolver to use.
    Resolver *Resolver

    // Cancel is an optional channel whose closure indicates that
    // the dial should be canceled. Not all types of dials support
    // cancellation.
    //
    // Deprecated: Use DialContext instead.
    Cancel <-chan struct{}

    // If Control is not nil, it is called after creating the network
    // connection but before actually dialing.
    //
    // Network and address parameters passed to Control method are not
    // necessarily the ones passed to Dial. For example, passing "tcp" to Dial
    // will cause the Control function to be called with "tcp4" or "tcp6".
    Control func(network, address string, c syscall.RawConn) error
}
```

A Dialer contains options for connecting to an address.

The zero value for each field is equivalent to dialing without that option. Dialing with the zero value of Dialer is therefore equivalent to just calling the Dial function.

It is safe to call Dialer's methods concurrently.

func (*Dialer) Dial

```
func (d *Dialer) Dial(network, address string) (Conn, error)
```

Dial connects to the address on the named network.

See func Dial for a description of the network and address parameters.

func (*Dialer) DialContext

```
func (d *Dialer) DialContext(ctx context.Context, network, address string) (Conn, err
```

DialContext connects to the address on the named network using the provided context.

The provided Context must be non-nil. If the context expires before the connection is complete, an error is returned. Once successfully connected, any expiration of the context will not affect the connection.

When using TCP, and the host in the address parameter resolves to multiple network addresses, any dial timeout (from d.Timeout or ctx) is spread over each consecutive dial, such that each is given an appropriate fraction of the time to connect. For example, if a host has 4 IP addresses and the timeout is 1 minute, the connect to each single address will be given 15 seconds to complete before trying the next one.

See func Dial for a description of the network and address parameters.

type Error

```
type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

An Error represents a network error.

type Flags

```
type Flags uint
```

```
const (
    FlagUp           Flags = 1 << iota // interface is up
    FlagBroadcast                      // interface supports broadcast access capabil
    FlagLoopback                       // interface is a loopback interface
    FlagPointToPoint                   // interface belongs to a point-to-point link
    FlagMulticast                      // interface supports multicast access capabil
)
```

func (Flags) String

```
func (f Flags) String() string
```

type HardwareAddr

```
type HardwareAddr []byte
```

A HardwareAddr represents a physical hardware address.

func ParseMAC

```
func ParseMAC(s string) (hw HardwareAddr, err error)
```

ParseMAC parses s as an IEEE 802 MAC-48, EUI-48, EUI-64, or a 20-octet IP over InfiniBand link-layer address using one of the following formats:

```
00:00:5e:00:53:01
02:00:5e:10:00:00:00:01
00:00:00:00:fe:80:00:00:00:00:00:00:02:00:5e:10:00:00:00:01
00-00-5e-00-53-01
02-00-5e-10-00-00-00-01
00-00-00-00-fe-80-00-00-00-00-00-00-02-00-5e-10-00-00-00-01
0000.5e00.5301
0200.5e10.0000.0001
0000.0000.fe80.0000.0000.0000.0200.5e10.0000.0001
```

func (HardwareAddr) String

```
func (a HardwareAddr) String() string
```

type IP

```
type IP []byte
```

An IP is a single IP address, a slice of bytes. Functions in this package accept either 4-byte (IPv4) or 16-byte (IPv6) slices as input.

Note that in this documentation, referring to an IP address as an IPv4 address or an IPv6 address is a semantic property of the address, not just the length of the byte slice: a 16-byte slice can still be an IPv4 address.

func IPv4

```
func IPv4(a, b, c, d byte) IP
```

IPv4 returns the IP address (in 16-byte form) of the IPv4 address a.b.c.d.

func LookupIP

```
func LookupIP(host string) ([]IP, error)
```

LookupIP looks up host using the local resolver. It returns a slice of that host's IPv4 and IPv6 addresses.

func ParseIP

```
func ParseIP(s string) IP
```

ParseIP parses s as an IP address, returning the result. The string s can be in IPv4 dotted decimal ("192.0.2.1"), IPv6 ("2001:db8::68"), or IPv4-mapped IPv6 ("::ffff:192.0.2.1") form. If s is not a valid textual representation of an IP address, ParseIP returns nil.

func (IP) DefaultMask

```
func (ip IP) DefaultMask() IPMask
```

DefaultMask returns the default IP mask for the IP address ip. Only IPv4 addresses have default masks; DefaultMask returns nil if ip is not a valid IPv4 address.

func (IP) Equal

```
func (ip IP) Equal(x IP) bool
```

Equal reports whether ip and x are the same IP address. An IPv4 address and that same address in IPv6 form are considered to be equal.

func (IP) IsGlobalUnicast

```
func (ip IP) IsGlobalUnicast() bool
```

IsGlobalUnicast reports whether ip is a global unicast address.

The identification of global unicast addresses uses address type identification as defined in RFC 1122, RFC 4632 and RFC 4291 with the exception of IPv4 directed broadcast addresses. It returns true even if ip is in IPv4 private address space or local IPv6 unicast address space.

func (IP) IsInterfaceLocalMulticast

```
func (ip IP) IsInterfaceLocalMulticast() bool
```

IsInterfaceLocalMulticast reports whether ip is an interface-local multicast address.

func (IP) IsLinkLocalMulticast

```
func (ip IP) IsLinkLocalMulticast() bool
```

IsLinkLocalMulticast reports whether ip is a link-local multicast address.

func (IP) IsLinkLocalUnicast

```
func (ip IP) IsLinkLocalUnicast() bool
```

IsLinkLocalUnicast reports whether ip is a link-local unicast address.

func (IP) IsLoopback

```
func (ip IP) IsLoopback() bool
```

IsLoopback reports whether ip is a loopback address.

func (IP) IsMulticast

```
func (ip IP) IsMulticast() bool
```

IsMulticast reports whether ip is a multicast address.

func (IP) IsUnspecified

```
func (ip IP) IsUnspecified() bool
```

IsUnspecified reports whether ip is an unspecified address, either the IPv4 address "0.0.0.0" or the IPv6 address "::".

func (IP) MarshalText

```
func (ip IP) MarshalText() ([]byte, error)
```

MarshalText implements the encoding.TextMarshaler interface. The encoding is the same as returned by String, with one exception: When len(ip) is zero, it returns an empty slice.

func (IP) Mask

```
func (ip IP) Mask(mask IPMask) IP
```

Mask returns the result of masking the IP address ip with mask.

func (IP) String

```
func (ip IP) String() string
```

String returns the string form of the IP address ip. It returns one of 4 forms:

```
- "<nil>", if ip has length 0
- dotted decimal ("192.0.2.1"), if ip is an IPv4 or IP4-mapped IPv6 address
- IPv6 ("2001:db8::1"), if ip is a valid IPv6 address
- the hexadecimal form of ip, without punctuation, if no other cases apply
```

func (IP) To16

```
func (ip IP) To16() IP
```

To16 converts the IP address ip to a 16-byte representation. If ip is not an IP address (it is the wrong length), To16 returns nil.

func (IP) To4

```
func (ip IP) To4() IP
```

To4 converts the IPv4 address ip to a 4-byte representation. If ip is not an IPv4 address, To4 returns nil.

func (*IP) UnmarshalText

```
func (ip *IP) UnmarshalText(text []byte) error
```

UnmarshalText implements the encoding.TextUnmarshaler interface. The IP address is expected in a form accepted by ParseIP.

type IPAddr

```
type IPAddr struct {
    IP   IP
    Zone string // IPv6 scoped addressing zone
}
```

IPAddr represents the address of an IP end point.

func ResolveIPAddr

```
func ResolveIPAddr(network, address string) (*IPAddr, error)
```

ResolveIPAddr returns an address of IP end point.

The network must be an IP network name.

If the host in the address parameter is not a literal IP address, ResolveIPAddr resolves the address to an address of IP end point. Otherwise, it parses the address as a literal IP address. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See func Dial for a description of the network and address parameters.

func (*IPAddr) Network

```
func (a *IPAddr) Network() string
```

Network returns the address's network name, "ip".

func (*IPAddr) String

```
func (a *IPAddr) String() string
```

type IPConn

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

IPConn is the implementation of the Conn and PacketConn interfaces for IP network connections.

func DialIP

```
func DialIP(network string, laddr, raddr *IPAddr) (*IPConn, error)
```

DialIP acts like Dial for IP networks.

The network must be an IP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

func ListenIP

```
func ListenIP(network string, laddr *IPAddr) (*IPConn, error)
```

ListenIP acts like ListenPacket for IP networks.

The network must be an IP network name; see func Dial for details.

If the IP field of laddr is nil or an unspecified IP address, ListenIP listens on all available IP addresses of the local system except multicast IP addresses.

func (*IPConn) Close

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

Close closes the connection.

func (*IPConn) File

```
func (c *IPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*IPConn) LocalAddr

```
func (c *IPConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

func (*IPConn) Read

```
func (c *IPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*IPConn) ReadFrom

```
func (c *IPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*IPConn) ReadFromIP

```
func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
```

ReadFromIP acts like ReadFrom but returns an IPAddr.

func (*IPConn) ReadMsgIP

```
func (c *IPConn) ReadMsgIP(b, oob []byte) (n, oobn, flags int, addr *IPAddr, err erro
```

ReadMsgIP reads a message from c, copying the payload into b and the associated out-of-band data into oob. It returns the number of bytes copied into b, the number of bytes copied into oob, the flags that were set on the message and the source address of the message.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

func (*IPConn) RemoteAddr

```
func (c *IPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

func (*IPConn) SetDeadline

```
func (c *IPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*IPConn) SetReadBuffer

```
func (c *IPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*IPConn) SetReadDeadline

```
func (c *IPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*IPConn) SetWriteBuffer

```
func (c *IPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*IPConn) SetWriteDeadline

```
func (c *IPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*IPConn) SyscallConn

```
func (c *IPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

func (*IPConn) Write

```
func (c *IPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

func (*IPConn) WriteMsgIP

```
func (c *IPConn) WriteMsgIP(b, oob []byte, addr *IPAddr) (n, oobn int, err error)
```

WriteMsgIP writes a message to addr via c, copying the payload from b and the associated out-of-band data from oob. It returns the number of payload and out-of-band bytes written.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

func (*IPConn) WriteTo

```
func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

func (*IPConn) WriteToIP

```
func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, error)
```

WriteToIP acts like WriteTo but takes an IPAddr.

type IPMask

```
type IPMask []byte
```

An IPMask is a bitmask that can be used to manipulate IP addresses for IP addressing and routing.

See type IPNet and func ParseCIDR for details.

func CIDRMask

```
func CIDRMask(ones, bits int) IPMask
```

CIDRMask returns an IPMask consisting of 'ones' 1 bits followed by 0s up to a total length of 'bits' bits. For a mask of this form, CIDRMask is the inverse of IPMask.Size.

func IPv4Mask

```
func IPv4Mask(a, b, c, d byte) IPMask
```

IPv4Mask returns the IP mask (in 4-byte form) of the IPv4 mask a.b.c.d.

func (IPMask) Size

```
func (m IPMask) Size() (ones, bits int)
```

Size returns the number of leading ones and total bits in the mask. If the mask is not in the canonical form--ones followed by zeros--then Size returns 0, 0.

func (IPMask) String

```
func (m IPMask) String() string
```

String returns the hexadecimal form of m, with no punctuation.

type IPNet

```
type IPNet struct {
    IP   IP     // network number
    Mask IPMask // network mask
}
```

An IPNet represents an IP network.

func (*IPNet) Contains

```
func (n *IPNet) Contains(ip IP) bool
```

Contains reports whether the network includes ip.

func (*IPNet) Network

```
func (n *IPNet) Network() string
```

Network returns the address's network name, "ip+net".

func (*IPNet) String

```
func (n *IPNet) String() string
```

String returns the CIDR notation of n like "192.0.2.0/24" or "2001:db8::/48" as defined in RFC 4632 and RFC 4291. If the mask is not in the canonical form, it returns the string which consists of an IP address, followed by a slash character and a mask expressed as hexadecimal form with no punctuation like "198.51.100.0/c000ff00".

type Interface

```
type Interface struct {
    Index        int          // positive integer that starts at one, zero is never u
    MTU          int          // maximum transmission unit
    Name         string       // e.g., "en0", "lo0", "eth0.100"
    HardwareAddr HardwareAddr // IEEE MAC-48, EUI-48 and EUI-64 form
    Flags        Flags        // e.g., FlagUp, FlagLoopback, FlagMulticast
}
```

Interface represents a mapping between network interface name and index. It also represents network interface facility information.

func InterfaceByIndex

```
func InterfaceByIndex(index int) (*Interface, error)
```

InterfaceByIndex returns the interface specified by index.

On Solaris, it returns one of the logical network interfaces sharing the logical data link; for more precision use InterfaceByName.

func InterfaceByName

```
func InterfaceByName(name string) (*Interface, error)
```

InterfaceByName returns the interface specified by name.

func Interfaces

```
func Interfaces() ([]Interface, error)
```

Interfaces returns a list of the system's network interfaces.

func (*Interface) Addrs

```
func (ifi *Interface) Addrs() ([]Addr, error)
```

Addrs returns a list of unicast interface addresses for a specific interface.

func (*Interface) MulticastAddrs

```
func (ifi *Interface) MulticastAddrs() ([]Addr, error)
```

MulticastAddrs returns a list of multicast, joined group addresses for a specific interface.

type InvalidAddrError

```
type InvalidAddrError string
```

func (InvalidAddrError) Error

```
func (e InvalidAddrError) Error() string
```

func (InvalidAddrError) Temporary

```
func (e InvalidAddrError) Temporary() bool
```

func (InvalidAddrError) Timeout

```
func (e InvalidAddrError) Timeout() bool
```

type ListenConfig

```
type ListenConfig struct {
    // If Control is not nil, it is called after creating the network
    // connection but before binding it to the operating system.
    //
    // Network and address parameters passed to Control method are not
    // necessarily the ones passed to Listen. For example, passing "tcp" to
    // Listen will cause the Control function to be called with "tcp4" or "tcp6".
    Control func(network, address string, c syscall.RawConn) error

    // KeepAlive specifies the keep-alive period for network
    // connections accepted by this listener.
    // If zero, keep-alives are enabled if supported by the protocol
    // and operating system. Network protocols or operating systems
    // that do not support keep-alives ignore this field.
    // If negative, keep-alives are disabled.
    KeepAlive time.Duration
}
```

ListenConfig contains options for listening to an address.

func (*ListenConfig) Listen

```
func (lc *ListenConfig) Listen(ctx context.Context, network, address string) (Listene
```

Listen announces on the local network address.

See func Listen for a description of the network and address parameters.

func (*ListenConfig) ListenPacket

```
func (lc *ListenConfig) ListenPacket(ctx context.Context, network, address string) (P
```

ListenPacket announces on the local network address.

See func ListenPacket for a description of the network and address parameters.

type Listener

```
type Listener interface {
    // Accept waits for and returns the next connection to the listener.
```

```
    Accept() (Conn, error)

    // Close closes the listener.
    // Any blocked Accept operations will be unblocked and return errors.
    Close() error

    // Addr returns the listener's network address.
    Addr() Addr
}
```

A Listener is a generic network listener for stream-oriented protocols.

Multiple goroutines may invoke methods on a Listener simultaneously.

func FileListener

```
func FileListener(f *os.File) (ln Listener, err error)
```

FileListener returns a copy of the network listener corresponding to the open file f. It is the caller's responsibility to close ln when finished. Closing ln does not affect f, and closing f does not affect ln.

func Listen

```
func Listen(network, address string) (Listener, error)
```

Listen announces on the local network address.

The network must be "tcp", "tcp4", "tcp6", "unix" or "unixpacket".

For TCP networks, if the host in the address parameter is empty or a literal unspecified IP address, Listen listens on all available unicast and anycast IP addresses of the local system. To only use IPv4, use network "tcp4". The address can use a host name, but this is not recommended, because it will create a listener for at most one of the host's IP addresses. If the port in the address parameter is empty or "0", as in "127.0.0.1:" or "[::1]:0", a port number is automatically chosen. The Addr method of Listener can be used to discover the chosen port.

See func Dial for a description of the network and address parameters.

type MX

```
type MX struct {
    Host string
    Pref uint16
}
```

An MX represents a single DNS MX record.

func LookupMX

```
func LookupMX(name string) ([]*MX, error)
```

LookupMX returns the DNS MX records for the given domain name sorted by preference.

type NS

```
type NS struct {
    Host string
}
```

An NS represents a single DNS NS record.

func LookupNS

```
func LookupNS(name string) ([]*NS, error)
```

LookupNS returns the DNS NS records for the given domain name.

type OpError

```
type OpError struct {
    // Op is the operation which caused the error, such as
    // "read" or "write".
    Op string

    // Net is the network type on which this error occurred,
    // such as "tcp" or "udp6".
    Net string

    // For operations involving a remote network connection, like
    // Dial, Read, or Write, Source is the corresponding local
    // network address.
    Source Addr

    // Addr is the network address for which this error occurred.
    // For local operations, like Listen or SetDeadline, Addr is
    // the address of the local endpoint being manipulated.
    // For operations involving a remote network connection, like
    // Dial, Read, or Write, Addr is the remote address of that
    // connection.
    Addr Addr

    // Err is the error that occurred during the operation.
    // The Error method panics if the error is nil.
    Err error
}
```

OpError is the error type usually returned by functions in the net package. It describes the operation, network type, and address of an error.

func (*OpError) Error

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

func (*OpError) Temporary

```
func (e *OpError) Temporary() bool
```

func (*OpError) Timeout

```
func (e *OpError) Timeout() bool
```

func (*OpError) Unwrap

```
func (e *OpError) Unwrap() error
```

type PacketConn

```
type PacketConn interface {
    // ReadFrom reads a packet from the connection,
    // copying the payload into p. It returns the number of
    // bytes copied into p and the return address that
    // was on the packet.
    // It returns the number of bytes read (0 <= n <= len(p))
    // and any error encountered. Callers should always process
    // the n > 0 bytes returned before considering the error err.
    // ReadFrom can be made to time out and return an error after a
    // fixed time limit; see SetDeadline and SetReadDeadline.
    ReadFrom(p []byte) (n int, addr Addr, err error)

    // WriteTo writes a packet with payload p to addr.
    // WriteTo can be made to time out and return an Error after a
    // fixed time limit; see SetDeadline and SetWriteDeadline.
    // On packet-oriented connections, write timeouts are rare.
    WriteTo(p []byte, addr Addr) (n int, err error)

    // Close closes the connection.
    // Any blocked ReadFrom or WriteTo operations will be unblocked and return errors
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // SetDeadline sets the read and write deadlines associated
    // with the connection. It is equivalent to calling both
    // SetReadDeadline and SetWriteDeadline.
    //
    // A deadline is an absolute time after which I/O operations
    // fail instead of blocking. The deadline applies to all future
```

```
    // and pending I/O, not just the immediately following call to
    // Read or Write. After a deadline has been exceeded, the
    // connection can be refreshed by setting a deadline in the future.
    //
    // If the deadline is exceeded a call to Read or Write or to other
    // I/O methods will return an error that wraps os.ErrDeadlineExceeded.
    // This can be tested using errors.Is(err, os.ErrDeadlineExceeded).
    // The error's Timeout method will return true, but note that there
    // are other possible errors for which the Timeout method will
    // return true even if the deadline has not been exceeded.
    //
    // An idle timeout can be implemented by repeatedly extending
    // the deadline after successful ReadFrom or WriteTo calls.
    //
    // A zero value for t means I/O operations will not time out.
    SetDeadline(t time.Time) error

    // SetReadDeadline sets the deadline for future ReadFrom calls
    // and any currently-blocked ReadFrom call.
    // A zero value for t means ReadFrom will not time out.
    SetReadDeadline(t time.Time) error

    // SetWriteDeadline sets the deadline for future WriteTo calls
    // and any currently-blocked WriteTo call.
    // Even if write times out, it may return n > 0, indicating that
    // some of the data was successfully written.
    // A zero value for t means WriteTo will not time out.
    SetWriteDeadline(t time.Time) error
}
```

PacketConn is a generic packet-oriented network connection.

Multiple goroutines may invoke methods on a PacketConn simultaneously.

func FilePacketConn

```
func FilePacketConn(f *os.File) (c PacketConn, err error)
```

FilePacketConn returns a copy of the packet network connection corresponding to the open file f. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

func ListenPacket

```
func ListenPacket(network, address string) (PacketConn, error)
```

ListenPacket announces on the local network address.

The network must be "udp", "udp4", "udp6", "unixgram", or an IP transport. The IP transports are "ip", "ip4", or "ip6" followed by a colon and a literal protocol number or a protocol name, as in "ip:1" or

"ip:icmp".

For UDP and IP networks, if the host in the address parameter is empty or a literal unspecified IP address, ListenPacket listens on all available IP addresses of the local system except multicast IP addresses. To only use IPv4, use network "udp4" or "ip4:proto". The address can use a host name, but this is not recommended, because it will create a listener for at most one of the host's IP addresses. If the port in the address parameter is empty or "0", as in "127.0.0.1:" or "[::1]:0", a port number is automatically chosen. The LocalAddr method of PacketConn can be used to discover the chosen port.

See func Dial for a description of the network and address parameters.

type ParseError

```
type ParseError struct {
    // Type is the type of string that was expected, such as
    // "IP address", "CIDR address".
    Type string

    // Text is the malformed text string.
    Text string
}
```

A ParseError is the error type of literal network address parsers.

func (*ParseError) Error

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

type Resolver

```
type Resolver struct {
    // PreferGo controls whether Go's built-in DNS resolver is preferred
    // on platforms where it's available. It is equivalent to setting
    // GODEBUG=netdns=go, but scoped to just this resolver.
    PreferGo bool

    // StrictErrors controls the behavior of temporary errors
    // (including timeout, socket errors, and SERVFAIL) when using
    // Go's built-in resolver. For a query composed of multiple
    // sub-queries (such as an A+AAAA address lookup, or walking the
    // DNS search list), this option causes such errors to abort the
    // whole query instead of returning a partial result. This is
    // not enabled by default because it may affect compatibility
    // with resolvers that process AAAA queries incorrectly.
    StrictErrors bool

    // Dial optionally specifies an alternate dialer for use by
    // Go's built-in DNS resolver to make TCP and UDP connections
    // to DNS services. The host in the address parameter will
    // always be a literal IP address and not a host name, and the
```

```
    // port in the address parameter will be a literal port number
    // and not a service name.
    // If the Conn returned is also a PacketConn, sent and received DNS
    // messages must adhere to RFC 1035 section 4.2.1, "UDP usage".
    // Otherwise, DNS messages transmitted over Conn must adhere
    // to RFC 7766 section 5, "Transport Protocol Selection".
    // If nil, the default dialer is used.
    Dial func(ctx context.Context, network, address string) (Conn, error)
    // contains filtered or unexported fields
}
```

A Resolver looks up names and numbers.

A nil *Resolver is equivalent to a zero Resolver.

func (*Resolver) LookupAddr

```
func (r *Resolver) LookupAddr(ctx context.Context, addr string) (names []string, err
```

LookupAddr performs a reverse lookup for the given address, returning a list of names mapping to that address.

func (*Resolver) LookupCNAME

```
func (r *Resolver) LookupCNAME(ctx context.Context, host string) (cname string, err e
```

LookupCNAME returns the canonical name for the given host. Callers that do not care about the canonical name can call LookupHost or LookupIP directly; both take care of resolving the canonical name as part of the lookup.

A canonical name is the final name after following zero or more CNAME records. LookupCNAME does not return an error if host does not contain DNS "CNAME" records, as long as host resolves to address records.

func (*Resolver) LookupHost

```
func (r *Resolver) LookupHost(ctx context.Context, host string) (addrs []string, err
```

LookupHost looks up the given host using the local resolver. It returns a slice of that host's addresses.

func (*Resolver) LookupIP

```
func (r *Resolver) LookupIP(ctx context.Context, network, host string) ([]IP, error)
```

LookupIP looks up host for the given network using the local resolver. It returns a slice of that host's IP addresses of the type specified by network. network must be one of "ip", "ip4" or "ip6".

func (*Resolver) LookupIPAddr

```
func (r *Resolver) LookupIPAddr(ctx context.Context, host string) ([]IPAddr, error)
```

LookupIPAddr looks up host using the local resolver. It returns a slice of that host's IPv4 and IPv6 addresses.

func (*Resolver) LookupMX

```
func (r *Resolver) LookupMX(ctx context.Context, name string) ([]*MX, error)
```

LookupMX returns the DNS MX records for the given domain name sorted by preference.

func (*Resolver) LookupNS

```
func (r *Resolver) LookupNS(ctx context.Context, name string) ([]*NS, error)
```

LookupNS returns the DNS NS records for the given domain name.

func (*Resolver) LookupPort

```
func (r *Resolver) LookupPort(ctx context.Context, network, service string) (port int
```

LookupPort looks up the port for the given network and service.

func (*Resolver) LookupSRV

```
func (r *Resolver) LookupSRV(ctx context.Context, service, proto, name string) (cname
```

LookupSRV tries to resolve an SRV query of the given service, protocol, and domain name. The proto is "tcp" or "udp". The returned records are sorted by priority and randomized by weight within a priority.

LookupSRV constructs the DNS name to look up following RFC 2782. That is, it looks up _service._proto.name. To accommodate services publishing SRV records under non-standard names, if both service and proto are empty strings, LookupSRV looks up name directly.

func (*Resolver) LookupTXT

```
func (r *Resolver) LookupTXT(ctx context.Context, name string) ([]string, error)
```

LookupTXT returns the DNS TXT records for the given domain name.

type SRV

```
type SRV struct {
    Target   string
```

```
    Port     uint16
    Priority uint16
    Weight   uint16
}
```

An SRV represents a single DNS SRV record.

func LookupSRV

```
func LookupSRV(service, proto, name string) (cname string, addrs []*SRV, err error)
```

LookupSRV tries to resolve an SRV query of the given service, protocol, and domain name. The proto is "tcp" or "udp". The returned records are sorted by priority and randomized by weight within a priority.

LookupSRV constructs the DNS name to look up following RFC 2782. That is, it looks up _service._proto.name. To accommodate services publishing SRV records under non-standard names, if both service and proto are empty strings, LookupSRV looks up name directly.

type TCPAddr

```
type TCPAddr struct {
    IP   IP
    Port int
    Zone string // IPv6 scoped addressing zone
}
```

TCPAddr represents the address of a TCP end point.

func ResolveTCPAddr

```
func ResolveTCPAddr(network, address string) (*TCPAddr, error)
```

ResolveTCPAddr returns an address of TCP end point.

The network must be a TCP network name.

If the host in the address parameter is not a literal IP address or the port is not a literal port number, ResolveTCPAddr resolves the address to an address of TCP end point. Otherwise, it parses the address as a pair of literal IP address and port number. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See func Dial for a description of the network and address parameters.

func (*TCPAddr) Network

```
func (a *TCPAddr) Network() string
```

Network returns the address's network name, "tcp".

func (*TCPAddr) String

```
func (a *TCPAddr) String() string
```

type TCPConn

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

TCPConn is an implementation of the Conn interface for TCP network connections.

func DialTCP

```
func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

DialTCP acts like Dial for TCP networks.

The network must be a TCP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

func (*TCPConn) Close

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

Close closes the connection.

func (*TCPConn) CloseRead

```
func (c *TCPConn) CloseRead() error
```

CloseRead shuts down the reading side of the TCP connection. Most callers should just use Close.

func (*TCPConn) CloseWrite

```
func (c *TCPConn) CloseWrite() error
```

CloseWrite shuts down the writing side of the TCP connection. Most callers should just use Close.

func (*TCPConn) File

```
func (c *TCPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*TCPConn) LocalAddr

```
func (c *TCPConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

func (*TCPConn) Read

```
func (c *TCPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*TCPConn) ReadFrom

```
func (c *TCPConn) ReadFrom(r io.Reader) (int64, error)
```

ReadFrom implements the io.ReaderFrom ReadFrom method.

func (*TCPConn) RemoteAddr

```
func (c *TCPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

func (*TCPConn) SetDeadline

```
func (c *TCPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*TCPConn) SetKeepAlive

```
func (c *TCPConn) SetKeepAlive(keepalive bool) error
```

SetKeepAlive sets whether the operating system should send keep-alive messages on the connection.

func (*TCPConn) SetKeepAlivePeriod

```
func (c *TCPConn) SetKeepAlivePeriod(d time.Duration) error
```

SetKeepAlivePeriod sets period between keep-alives.

func (*TCPConn) SetLinger

```
func (c *TCPConn) SetLinger(sec int) error
```

SetLinger sets the behavior of Close on a connection which still has data waiting to be sent or to be acknowledged.

If sec < 0 (the default), the operating system finishes sending the data in the background.

If sec == 0, the operating system discards any unsent or unacknowledged data.

If sec > 0, the data is sent in the background as with sec < 0. On some operating systems after sec seconds have elapsed any remaining unsent data may be discarded.

func (*TCPConn) SetNoDelay

```
func (c *TCPConn) SetNoDelay(noDelay bool) error
```

SetNoDelay controls whether the operating system should delay packet transmission in hopes of sending fewer packets (Nagle's algorithm). The default is true (no delay), meaning that data is sent as soon as possible after a Write.

func (*TCPConn) SetReadBuffer

```
func (c *TCPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*TCPConn) SetReadDeadline

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*TCPConn) SetWriteBuffer

```
func (c *TCPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*TCPConn) SetWriteDeadline

```
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*TCPConn) SyscallConn

```
func (c *TCPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

func (*TCPConn) Write

```
func (c *TCPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

type TCPListener

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

TCPListener is a TCP network listener. Clients should typically use variables of type Listener instead of assuming TCP.

func ListenTCP

```
func ListenTCP(network string, laddr *TCPAddr) (*TCPListener, error)
```

ListenTCP acts like Listen for TCP networks.

The network must be a TCP network name; see func Dial for details.

If the IP field of laddr is nil or an unspecified IP address, ListenTCP listens on all available unicast and anycast IP addresses of the local system. If the Port field of laddr is 0, a port number is automatically chosen.

func (*TCPListener) Accept

```
func (l *TCPListener) Accept() (Conn, error)
```

Accept implements the Accept method in the Listener interface; it waits for the next call and returns a generic Conn.

func (*TCPListener) AcceptTCP

```
func (l *TCPListener) AcceptTCP() (*TCPConn, error)
```

AcceptTCP accepts the next incoming call and returns the new connection.

func (*TCPListener) Addr

```
func (l *TCPListener) Addr() Addr
```

Addr returns the listener's network address, a *TCPAddr. The Addr returned is shared by all invocations of Addr, so do not modify it.

func (*TCPListener) Close

```
func (l *TCPListener) Close() error
```

Close stops listening on the TCP address. Already Accepted connections are not closed.

func (*TCPListener) File

```
func (l *TCPListener) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing l does not affect f, and closing f does not affect l.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*TCPListener) SetDeadline

```
func (l *TCPListener) SetDeadline(t time.Time) error
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

func (*TCPListener) SyscallConn

```
func (l *TCPListener) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

The returned RawConn only supports calling Control. Read and Write return an error.

type UDPAddr

```
type UDPAddr struct {
    IP   IP
    Port int
```

```
    Zone string // IPv6 scoped addressing zone
}
```

UDPAddr represents the address of a UDP end point.

func ResolveUDPAddr

```
func ResolveUDPAddr(network, address string) (*UDPAddr, error)
```

ResolveUDPAddr returns an address of UDP end point.

The network must be a UDP network name.

If the host in the address parameter is not a literal IP address or the port is not a literal port number, ResolveUDPAddr resolves the address to an address of UDP end point. Otherwise, it parses the address as a pair of literal IP address and port number. The address parameter can use a host name, but this is not recommended, because it will return at most one of the host name's IP addresses.

See func Dial for a description of the network and address parameters.

func (*UDPAddr) Network

```
func (a *UDPAddr) Network() string
```

Network returns the address's network name, "udp".

func (*UDPAddr) String

```
func (a *UDPAddr) String() string
```

type UDPConn

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

UDPConn is the implementation of the Conn and PacketConn interfaces for UDP network connections.

func DialUDP

```
func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialUDP acts like Dial for UDP networks.

The network must be a UDP network name; see func Dial for details.

If laddr is nil, a local address is automatically chosen. If the IP field of raddr is nil or an unspecified IP address, the local system is assumed.

func ListenMulticastUDP

```
func ListenMulticastUDP(network string, ifi *Interface, gaddr *UDPAddr) (*UDPConn, er
```

ListenMulticastUDP acts like ListenPacket for UDP networks but takes a group address on a specific network interface.

The network must be a UDP network name; see func Dial for details.

ListenMulticastUDP listens on all available IP addresses of the local system including the group, multicast IP address. If ifi is nil, ListenMulticastUDP uses the system-assigned multicast interface, although this is not recommended because the assignment depends on platforms and sometimes it might require routing configuration. If the Port field of gaddr is 0, a port number is automatically chosen.

ListenMulticastUDP is just for convenience of simple, small applications. There are golang.org/x/net/ipv4 and golang.org/x/net/ipv6 packages for general purpose uses.

func ListenUDP

```
func ListenUDP(network string, laddr *UDPAddr) (*UDPConn, error)
```

ListenUDP acts like ListenPacket for UDP networks.

The network must be a UDP network name; see func Dial for details.

If the IP field of laddr is nil or an unspecified IP address, ListenUDP listens on all available IP addresses of the local system except multicast IP addresses. If the Port field of laddr is 0, a port number is automatically chosen.

func (*UDPConn) Close

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

Close closes the connection.

func (*UDPConn) File

```
func (c *UDPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*UDPConn) LocalAddr

```
func (c *UDPConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

func (*UDPConn) Read

```
func (c *UDPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*UDPConn) ReadFrom

```
func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*UDPConn) ReadFromUDP

```
func (c *UDPConn) ReadFromUDP(b []byte) (int, *UDPAddr, error)
```

ReadFromUDP acts like ReadFrom but returns a UDPAddr.

func (*UDPConn) ReadMsgUDP

```
func (c *UDPConn) ReadMsgUDP(b, oob []byte) (n, oobn, flags int, addr *UDPAddr, err e
```

ReadMsgUDP reads a message from c, copying the payload into b and the associated out-of-band data into oob. It returns the number of bytes copied into b, the number of bytes copied into oob, the flags that were set on the message and the source address of the message.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

func (*UDPConn) RemoteAddr

```
func (c *UDPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

func (*UDPConn) SetDeadline

```
func (c *UDPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*UDPConn) SetReadBuffer

```
func (c *UDPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*UDPConn) SetReadDeadline

```
func (c *UDPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*UDPConn) SetWriteBuffer

```
func (c *UDPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*UDPConn) SetWriteDeadline

```
func (c *UDPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*UDPConn) SyscallConn

```
func (c *UDPConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

func (*UDPConn) Write

```
func (c *UDPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

func (*UDPConn) WriteMsgUDP

```
func (c *UDPConn) WriteMsgUDP(b, oob []byte, addr *UDPAddr) (n, oobn int, err error)
```

WriteMsgUDP writes a message to addr via c if c isn't connected, or to c's remote address if c is connected (in which case addr must be nil). The payload is copied from b and the associated out-of-band data is copied from oob. It returns the number of payload and out-of-band bytes written.

The packages golang.org/x/net/ipv4 and golang.org/x/net/ipv6 can be used to manipulate IP-level socket options in oob.

func (*UDPConn) WriteTo

```
func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

func (*UDPConn) WriteToUDP

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP acts like WriteTo but takes a UDPAddr.

type UnixAddr

```
type UnixAddr struct {
    Name string
    Net  string
}
```

UnixAddr represents the address of a Unix domain socket end point.

func ResolveUnixAddr

```
func ResolveUnixAddr(network, address string) (*UnixAddr, error)
```

ResolveUnixAddr returns an address of Unix domain socket end point.

The network must be a Unix network name.

See func Dial for a description of the network and address parameters.

func (*UnixAddr) Network

```
func (a *UnixAddr) Network() string
```

Network returns the address's network name, "unix", "unixgram" or "unixpacket".

func (*UnixAddr) String

```
func (a *UnixAddr) String() string
```

type UnixConn

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

UnixConn is an implementation of the Conn interface for connections to Unix domain sockets.

func DialUnix

```
func DialUnix(network string, laddr, raddr *UnixAddr) (*UnixConn, error)
```

DialUnix acts like Dial for Unix networks.

The network must be a Unix network name; see func Dial for details.

If laddr is non-nil, it is used as the local address for the connection.

func ListenUnixgram

```
func ListenUnixgram(network string, laddr *UnixAddr) (*UnixConn, error)
```

ListenUnixgram acts like ListenPacket for Unix networks.

The network must be "unixgram".

func (*UnixConn) Close

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

Close closes the connection.

func (*UnixConn) CloseRead

```
func (c *UnixConn) CloseRead() error
```

CloseRead shuts down the reading side of the Unix domain connection. Most callers should just use Close.

func (*UnixConn) CloseWrite

```
func (c *UnixConn) CloseWrite() error
```

CloseWrite shuts down the writing side of the Unix domain connection. Most callers should just use Close.

func (*UnixConn) File

```
func (c *UnixConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*UnixConn) LocalAddr

```
func (c *UnixConn) LocalAddr() Addr
```

LocalAddr returns the local network address. The Addr returned is shared by all invocations of LocalAddr, so do not modify it.

func (*UnixConn) Read

```
func (c *UnixConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*UnixConn) ReadFrom

```
func (c *UnixConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*UnixConn) ReadFromUnix

```
func (c *UnixConn) ReadFromUnix(b []byte) (int, *UnixAddr, error)
```

ReadFromUnix acts like ReadFrom but returns a UnixAddr.

func (*UnixConn) ReadMsgUnix

```
func (c *UnixConn) ReadMsgUnix(b, oob []byte) (n, oobn, flags int, addr *UnixAddr, er
```

ReadMsgUnix reads a message from c, copying the payload into b and the associated out-of-band data into oob. It returns the number of bytes copied into b, the number of bytes copied into oob, the flags that were set on the message and the source address of the message.

Note that if len(b) == 0 and len(oob) > 0, this function will still read (and discard) 1 byte from the connection.

func (*UnixConn) RemoteAddr

```
func (c *UnixConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address. The Addr returned is shared by all invocations of RemoteAddr, so do not modify it.

func (*UnixConn) SetDeadline

```
func (c *UnixConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*UnixConn) SetReadBuffer

```
func (c *UnixConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*UnixConn) SetReadDeadline

```
func (c *UnixConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*UnixConn) SetWriteBuffer

```
func (c *UnixConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*UnixConn) SetWriteDeadline

```
func (c *UnixConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*UnixConn) SyscallConn

```
func (c *UnixConn) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

func (*UnixConn) Write

```
func (c *UnixConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

func (*UnixConn) WriteMsgUnix

```
func (c *UnixConn) WriteMsgUnix(b, oob []byte, addr *UnixAddr) (n, oobn int, err erro
```

WriteMsgUnix writes a message to addr via c, copying the payload from b and the associated out-of-band data from oob. It returns the number of payload and out-of-band bytes written.

Note that if len(b) == 0 and len(oob) > 0, this function will still write 1 byte to the connection.

func (*UnixConn) WriteTo

```
func (c *UnixConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

func (*UnixConn) WriteToUnix

```
func (c *UnixConn) WriteToUnix(b []byte, addr *UnixAddr) (int, error)
```

WriteToUnix acts like WriteTo but takes a UnixAddr.

type UnixListener

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

UnixListener is a Unix domain socket listener. Clients should typically use variables of type Listener instead of assuming Unix domain sockets.

func ListenUnix

```
func ListenUnix(network string, laddr *UnixAddr) (*UnixListener, error)
```

ListenUnix acts like Listen for Unix networks.

The network must be "unix" or "unixpacket".

func (*UnixListener) Accept

```
func (l *UnixListener) Accept() (Conn, error)
```

Accept implements the Accept method in the Listener interface. Returned connections will be of type *UnixConn.

func (*UnixListener) AcceptUnix

```
func (l *UnixListener) AcceptUnix() (*UnixConn, error)
```

AcceptUnix accepts the next incoming call and returns the new connection.

func (*UnixListener) Addr

```
func (l *UnixListener) Addr() Addr
```

Addr returns the listener's network address. The Addr returned is shared by all invocations of Addr, so do not modify it.

func (*UnixListener) Close

```
func (l *UnixListener) Close() error
```

Close stops listening on the Unix address. Already accepted connections are not closed.

func (*UnixListener) File

```
func (l *UnixListener) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File. It is the caller's responsibility to close f when finished. Closing l does not affect f, and closing f does not affect l.

The returned os.File's file descriptor is different from the connection's. Attempting to change properties of the original using this duplicate may or may not have the desired effect.

func (*UnixListener) SetDeadline

```
func (l *UnixListener) SetDeadline(t time.Time) error
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

func (*UnixListener) SetUnlinkOnClose

```
func (l *UnixListener) SetUnlinkOnClose(unlink bool)
```

SetUnlinkOnClose sets whether the underlying socket file should be removed from the file system when the listener is closed.

The default behavior is to unlink the socket file only when package net created it. That is, when the listener and the underlying socket file were created by a call to Listen or ListenUnix, then by default closing the listener will remove the socket file. but if the listener was created by a call to FileListener to use an already existing socket file, then by default closing the listener will not remove the socket file.

func (*UnixListener) SyscallConn

```
func (l *UnixListener) SyscallConn() (syscall.RawConn, error)
```

SyscallConn returns a raw network connection. This implements the syscall.Conn interface.

The returned RawConn only supports calling Control. Read and Write return an error.

type UnknownNetworkError

```
type UnknownNetworkError string
```

func (UnknownNetworkError) Error

```
func (e UnknownNetworkError) Error() string
```

func (UnknownNetworkError) Temporary

```
func (e UnknownNetworkError) Temporary() bool
```

func (UnknownNetworkError) Timeout

```
func (e UnknownNetworkError) Timeout() bool
```

BUGs

- On JS and Windows, the FileConn, FileListener and FilePacketConn functions are not implemented.
- On JS, methods and functions related to Interface are not implemented.
- On AIX, DragonFly BSD, NetBSD, OpenBSD, Plan 9 and Solaris, the MulticastAddrs method of Interface is not implemented.
- On every POSIX platform, reads from the "ip4" network using the ReadFrom or ReadFromIP method might not return a complete IPv4 packet, including its header, even if there is space available. This can occur even in cases where Read or ReadMsgIP could return a complete packet. For this reason, it is recommended that you do not use these methods if it is important to receive a full packet.

  The Go 1 compatibility guidelines make it impossible for us to change the behavior of these methods; use Read or ReadMsgIP instead.

- On JS and Plan 9, methods and functions related to IPConn are not implemented.
- On Windows, the File method of IPConn is not implemented.
- On DragonFly BSD and OpenBSD, listening on the "tcp" and "udp" networks does not listen for both IPv4 and IPv6 connections. This is due to the fact that IPv4 traffic will not be routed to an IPv6 socket - two separate sockets are required if both address families are to be supported. See inet6(4) for details.
- On Windows, the Write method of syscall.RawConn does not integrate with the runtime's network poller. It cannot wait for the connection to become writeable, and does not respect deadlines. If the user-provided callback returns false, the Write method will fail immediately.
- On JS and Plan 9, the Control, Read and Write methods of syscall.RawConn are not implemented.
- On JS and Windows, the File method of TCPConn and TCPListener is not implemented.
- On Plan 9, the ReadMsgUDP and WriteMsgUDP methods of UDPConn are not implemented.
- On Windows, the File method of UDPConn is not implemented.
- On JS, methods and functions related to UDPConn are not implemented.
- On JS and Plan 9, methods and functions related to UnixConn and UnixListener are not implemented.
- On Windows, methods and functions related to UnixConn and UnixListener don't work for "unixgram" and "unixpacket".