Go: Deep dive into net package learning from TCP server

#go#webdev

FYI: A previous title was "How Go handles network and system calls when TCP server".

Key takeaways

With the net package, you can create a TCP server with such simple and little code (The Go playground).
If the host in the address parameter is empty or a literal unspecified IP address, listen on all available unicast and anycast IP addresses of the local system.
There are two kinds of DNS resolver: Go's built-in DNS resolver and uses Cgo DNS resolver. By default, Go's built-in DNS resolver is used.
If the port in the address parameter is empty or "0", a port number is automatically chosen by the system call bind.
Go accepts an incoming connection by using the system call accept.

TCP server in Go

By using packages in standard libraries, you make it easier to write and manage your code and ensure that it's reliable. They are well designed and provide more functionality than is normally found in traditional standard libraries. In particular, net package is often used by web application engineers, which provides a portable interface for network I/O, including TCP/IO, UDP, domain name resolution, and Unix domain sockets.
With the net package, you can create a TCP server with such simple and little code (The Go playground).

```
package main

import (
    "fmt"
    "net"
)

func main() {
    // Listen for incoming connections.
    addr := "localhost:8888"
    l, err := net.Listen("tcp", addr)
    if err != nil {
        panic(err)
    }
    defer l.Close()
    host, port, err := net.SplitHostPort(l.Addr().String())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Listening on host: %s, port: %s\n", host, port)

    for {
        // Listen for an incoming connection
        conn, err := l.Accept()
        if err != nil {
            panic(err)
        }
        // Handle connections in a new goroutine
        go func(conn net.Conn) {
            buf := make([]byte, 1024)
            len, err := conn.Read(buf)
            if err != nil {
                fmt.Printf("Error reading: %#v\n", err)
                return
            }
            fmt.Printf("Message received: %s\n", string(buf[:len]))

            conn.Write([]byte("Message received.\n"))
            conn.Close()
        }(conn)
    }
}
```

When you run this Go script, it will start a TCP server that listens for incoming connections on port 8888.

```
$ go run main.go

Listening on host: 127.0.0.1, port: 8888
```

If you make a TCP connection to this server with nc command, you will see that the server has successfully received the message.

```
$ echo -n "How's it going?" | nc localhost 8888
Message received.
```

The implementation looks very simple and easy but knowing what is doing in the net package makes it easier to deal with advanced issues. This article tries to deep dive into the design and implementation of the net package.

Note that the version of Go presented in this article is 1.17.5.

Listen for incoming connections

At the begging of the main function, use net.Listen to listen for incoming connections.

```
func main() {
    // Listen for incoming connections.
    addr := "localhost:8888"
    l, err := net.Listen("tcp", addr)
    if err != nil {
        panic(err)
    }
    defer l.Close()
    host, port, err := net.SplitHostPort(l.Addr().String())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Listening on host: %s, port: %s\n", host, port)
```

The signature of net.Listen is as follow.

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

Network types: "tcp", "tcp4", "tcp6"

The first argument "network" must be "tcp", "tcp4", "tcp6", "unix" or "unixpacket".
"tcp4" means IPv4 only, and "tcp6" means IPv6 only. "tcp" means IPv4 or IPv6 as the comment on net.ListenConfig mentions.
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".

Go and Cgo resolver

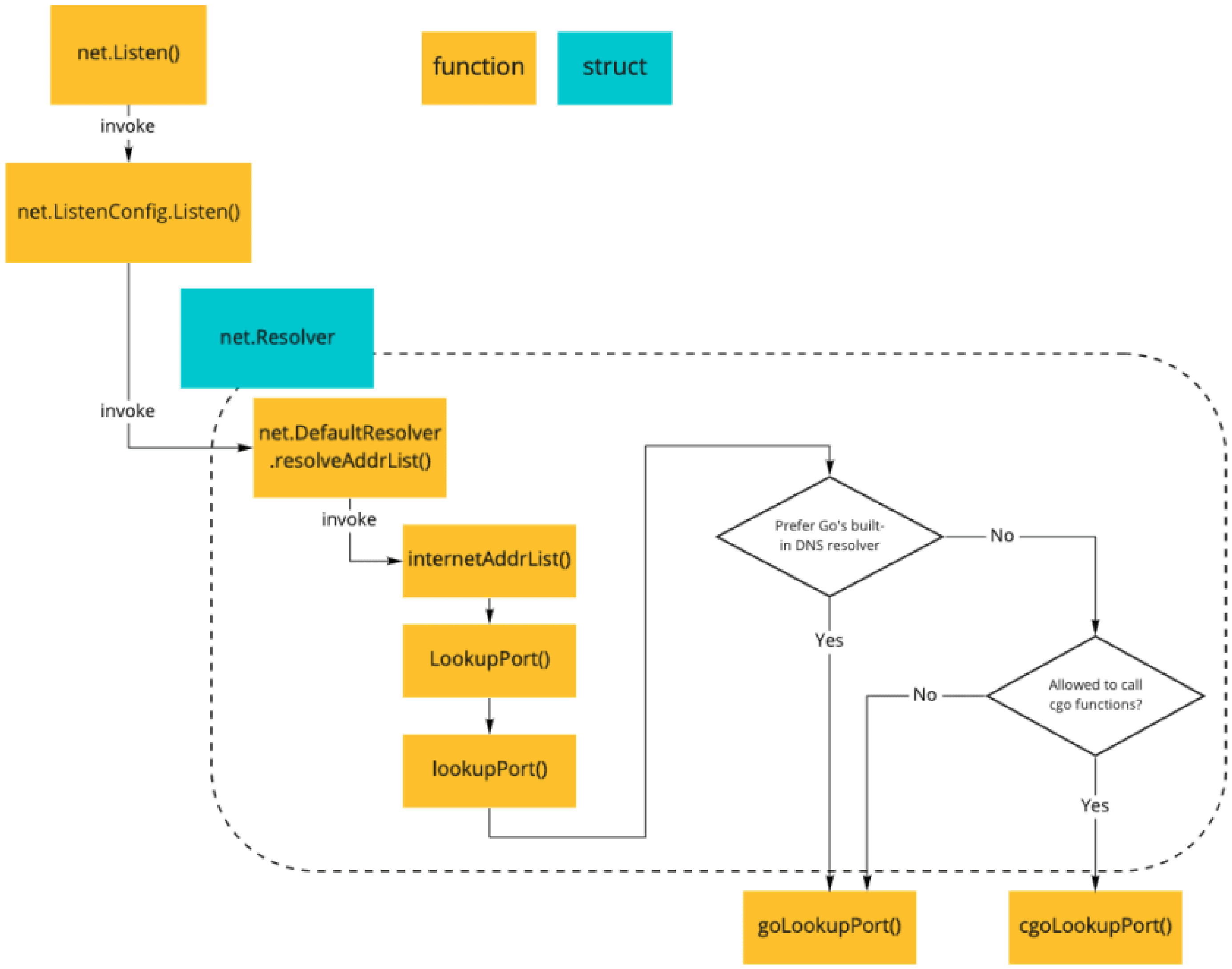
The second argument "address" means the local network address and there are several patterns for TCP networks.
If the host in the address parameter is empty or a literal unspecified IP address, listen on all available unicast and anycast IP addresses of the local system.
In the above example, I pass the literal unspecified IP address "localhost", and net.Listen casts as the IP "127.0.0.1".
For another example, if you pass an empty address as the following code:

```
// empty host
addr := ":8888"
l, err := net.Listen("tcp", addr)
if err != nil {
    panic(err)
}
defer l.Close()
host, port, err := net.SplitHostPort(l.Addr().String())
if err != nil {
    panic(err)
}
fmt.Printf("Listening on host: %s, port: %s\n", host, port)
```

The host and port where the server listens will be "::" and "8888".

```
$ go run main.go
Listening on host: ::, port: 8888
```

As described in the comment on net.Dial, if the host is empty or a literal unspecified IP address, for TCP and UDP ":80", "0.0.0.0:80" or "[::]:80.
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.
Let's take a look at the inner implementation of this behavior. In the net package, several functions are invoked in the order shown in the following figure.



net.Listen invokes ListenConfig.Listen that announces on the local network address.
Then it creates DefaultResolver which type is net.Resolver who actually resolves the network IP address, and then invokes Resolver.resolveAddrList.

```
type Resolver struct {
    PreferGo bool
    StrictErrors bool
    Dial func(ctx context.Context, network, address string) (Conn, error)
    // contains filtered or unexported fields
}
```

From there various functions are invoked, most notably Resolver.lookupPort decides whether it uses Go's built-in DNS resolver or uses Cgo DNS resolver.

```
func (r *Resolver) lookupPort(ctx context.Context, network, service string) (int, error) {
    if !r.preferGo() && systemConf().canUseCgo() {
        if port, err, ok := cgoLookupPort(ctx, network, service); ok {
            if err != nil {
                // Issue 18213: if cgo fails, first check to see whether we
                // have the answer baked-in to the net package.
                if port, err := goLookupPort(network, service); err == nil {
                    return port, nil
                }
            }
            return port, err
        }
    }
    return goLookupPort(network, service)
}
```

Go's built-in DNS resolver sends DNS requests directly to the server listed in `/etc/resolv.conf`, and Cgo-based resolver calls C library routines such as getaddrinfo and getnameinfo.
By default, Go's built-in DNS resolver is used because a blocked DNS request consumes only a goroutine, while a blocked C call consumes an operating system thread. See Name Resolution in the GoDoc of the net package for more detail.
Also, You can see a use case of the Cgo DNS resolver from DNS Resolution in Go and Cgo.

A way to dynamically choose a port

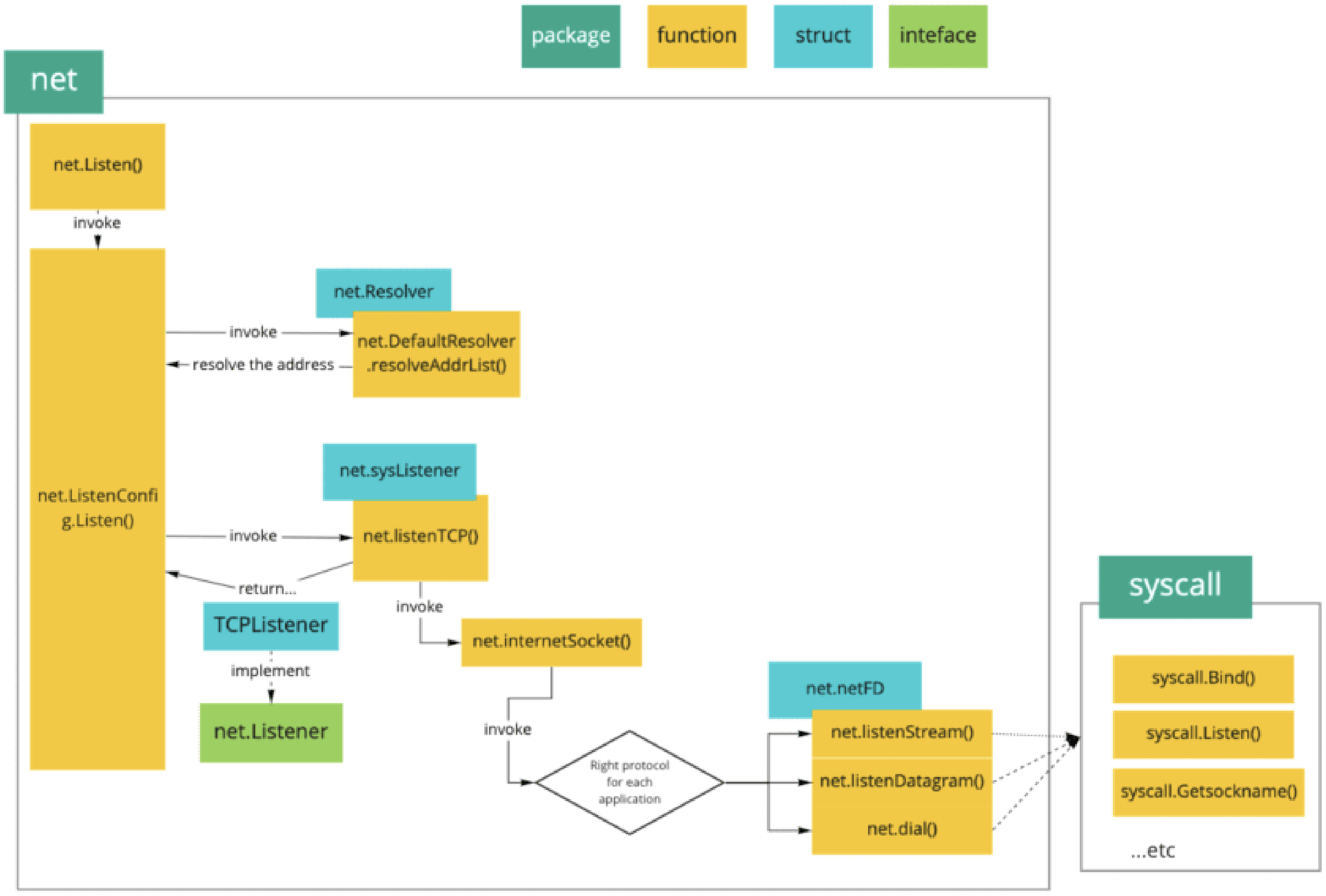
In addition, if the port in the address parameter is empty or "0", a port number is automatically chosen.

```
// a port number will be automatically chosen
addr := "127.0.0.1:"
l, err := net.Listen("tcp", addr)
if err != nil {
    panic(err)
}
defer l.Close()
host, port, err := net.SplitHostPort(l.Addr().String())
if err != nil {
    panic(err)
}
fmt.Printf("Listening on host: %s, port: %s\n", host, port)
```

For example, if you pass "127.0.0.1:" to the address parameter, a port number will be automatically chosen every time you run it.

```
$ go run main.go
Listening on host: 127.0.0.1, port: 51962

$ go run main.go
Listening on host: 127.0.0.1, port: 51978

$ go run main.go
Listening on host: 127.0.0.1, port: 51979
```

It's an interesting behavior, how is the port automatically selected? Let's take a look at the inner implementation of this behavior. Here is the overview of the net package's implementation.



As you can see, net.Listen invokes net.ListenConfig.Listen.
Then, net.Resolver.resolveAddrList resolves the address as "0" if the port in the address parameter is empty or "0". It is tested at TestLookupPort in the net package.

```
func TestLookupPort(t *testing.T) {
    // (omit)
    type test struct {
        network string
        name    string
        port    int
        ok      bool
    }
    var tests = []test{
        {"tcp", "0", 0, true},
        {"udp", "0", 0, true},
        {"udp", "domain", 53, true},

        {"--badnet--", "zzz", 0, false},
        {"tcp", "--badport--", 0, false},
        {"tcp", "-1", 0, false},
        {"tcp", "65536", 0, false},
        {"udp", "-1", 0, false},
        {"udp", "65536", 0, false},
        {"tcp", "123456789", 0, false},
        // Issue 13610: LookupPort("tcp", "")
        {"tcp", "", 0, true},
        {"tcp4", "", 0, true},
        {"tcp6", "", 0, true},
        {"udp", "", 0, true},
        {"udp4", "", 0, true},
        {"udp6", "", 0, true},
    }

    // (omit)

    for _, tt := range tests {
        port, err := LookupPort(tt.network, tt.name)
        if port != tt.port || (err == nil) != tt.ok {
            t.Errorf("LookupPort(%q, %q) = %d, %v; want %d, error=%t", tt.network, tt.name, port, err, tt.port, !tt.ok)
        }
        if err != nil {
            if perr := parseLookupPortError(err); perr != nil {
                t.Error(perr)
            }
        }
    }
}
```

Next, net.ListenConfig.Listen creates a net.sysListener that contains a Listen's parameters and configuration. When the network protocol is "tcp", net.ListenConfig.Listen invokes net.sysListener.listenTCP.

```
func (lc *ListenConfig) Listen(ctx context.Context, network, address string) (Listener, error) {
    addrs, err := DefaultResolver.resolveAddrList(ctx, "listen", network, address, nil)
    if err != nil {
        return nil, &OpError{Op: "listen", Net: network, Source: nil, Addr: nil, Err: err}
    }
    sl := &sysListener{
        ListenConfig: *lc,
        network:      network,
        address:      address,
    }
    var l Listener
    la := addrs.first(isIPv4)
    switch la := la.(type) {
    case *TCPAddr:
        l, err = sl.listenTCP(ctx, la)
    case *UnixAddr:
        l, err = sl.listenUnix(ctx, la)

    // (omit)
}
```

net.sysListener.listenTCP invokes net.internetSocket.

```
func (sl *sysListener) listenTCP(ctx context.Context, laddr *TCPAddr) (*TCPListener, error) {
    fd, err := internetSocket(ctx, sl.network, laddr, nil, syscall.SOCK_STREAM, 0, "listen", sl.ListenConfig.Control)
    if err != nil {
        return nil, err
    }
    return &TCPListener{fd: fd, lc: sl.ListenConfig}, nil
}
```

Then it invokes net.socket.

```
func socket(ctx context.Context, net string, family, sotype, proto int, ipv6only bool, laddr, raddr sockaddr, ctrlFn fur
    // (omit)

    if fd, err = newFD(s, family, sotype, net); err != nil {
        poll.CloseFunc(s)
        return nil, err
    }

    // (omit)
}
```

It returns a network file descriptor that is ready for asynchronous I/O using the network poller. The type representing a network file descriptor is net.netFD.

```
type netFD struct {
    pfd poll.FD

    family      int
    sotype      int
    isConnected bool
    net         string
    laddr       Addr
    raddr       Addr
}
```

pool.FD is the exported type representing a file descriptor which is used by the net and os packages. It is an internal package, so you can't use it directly.
Let's go back to the rest of net.socket.

```
func socket(ctx context.Context, net string, family, sotype, proto int, ipv6only bool, laddr, raddr sockaddr, ctrlFn fur

    // (omit)

    if laddr != nil && raddr == nil {
        switch sotype {
        case syscall.SOCK_STREAM, syscall.SOCK_SEQPACKET:
            if err := fd.listenStream(laddr, listenerBacklog(), ctrlFn); err != nil {
                fd.Close()
                return nil, err
            }
            return fd, nil
        case syscall.SOCK_DGRAM:
            if err := fd.listenDatagram(laddr, ctrlFn); err != nil {
```

```
            fd.Close()
            return nil, err
        }
        return fd, nil
    }
}
if err := fd.dial(ctx, laddr, raddr, ctrlFn); err != nil {
    fd.Close()
    return nil, err
}
return fd, nil
}
```

that all bytes received will be identical and in the same order as those sent. In contract, UDP (User Datagram Protocol) provides a connectionless datagram service that prioritizes time over reliability. That's why net.internetSocket chooses the socket type syscall.SOCK_STREAM that provides sequenced, reliable, two-way, connection-byte streams as Linux programmer's manual describes.

```
func (sl *sysListener) listenTCP(ctx context.Context, laddr *TCPAddr) (*TCPListener, error) {
    fd, err := internetSocket(ctx, sl.network, laddr, nil, syscall.SOCK_STREAM, 0, "listen", sl.ListenConfig.Control)
```

net.socket invokes net.netFD.listenStream when syscall.SOCK_STREAM is passed.

```
switch sotype {
case syscall.SOCK_STREAM, syscall.SOCK_SEQPACKET:
    if err := fd.listenStream(laddr, listenerBacklog(), ctrlFn);
```

Finally, we reached the last place.

```
func (fd *netFD) listenStream(laddr sockaddr, backlog int, ctrlFn func(string, string, syscall.RawConn) error) error {

    // (omit)

    if err = syscall.Bind(fd.pfd.Sysfd, lsa); err != nil {
        return os.NewSyscallError("bind", err)
    }
    if err = listenFunc(fd.pfd.Sysfd, backlog); err != nil {
        return os.NewSyscallError("listen", err)
    }
    if err = fd.init(); err != nil {
        return err
    }
    lsa, _ = syscall.Getsockname(fd.pfd.Sysfd)
    fd.setAddr(fd.addrFunc()(lsa), nil)
    return nil
}
```

net.netFD.listenStream executes three system calls: bind, listen, and getsockname.

bind assigns the socket address specified by given address to the socket referred to by the file descriptor.

```
if err = syscall.Bind(fd.pfd.Sysfd, lsa); err != nil {
```

listen marks the socket referred to by the file descriptor as a passive socket, that is, as a socket that will be used to accept incoming requests using accept

```
if err = listenFunc(fd.pfd.Sysfd, backlog); err != nil {
```

getsockname returns the current address to which the socket is bound. You can know the current address even when the port is randomly chosen.

```
lsa, _ = syscall.Getsockname(fd.pfd.Sysfd)
```

Let's go back to the first question: How is the port automatically selected? The answer is the specification of bind.
Ports that can be used by the TCP protocol are 16 bits in the protocol header so they can hold values between 0 - 65535.
The bind uses a zero in the port number to mean "pick random one" and bind its address.
For example, the specification of the bind function in Linux is described in ip(7) — Linux manual page.
An ephemeral port is allocated to a socket in the following circumstances:
the port number in a socket address is specified as 0 when calling bind(2);
Ephemeral port is a port of a transport layer protocol that is used for only a short period of time for the duration of a communication session. It is also called dynamic ports, because it is used on a per request basis. The IANA assigns the the range 49152-65535 for the selection of ephemeral ports as RFC 6056 describes.
The port range varies depending on the server system, but the same is true not only for Linux, but also other systems. For example, in Windows, bind function (winsock.h) assigns a unique port to the application from the dynamic client port range 49152 and 65535 (when on Windows Vista later).

Accept a connection

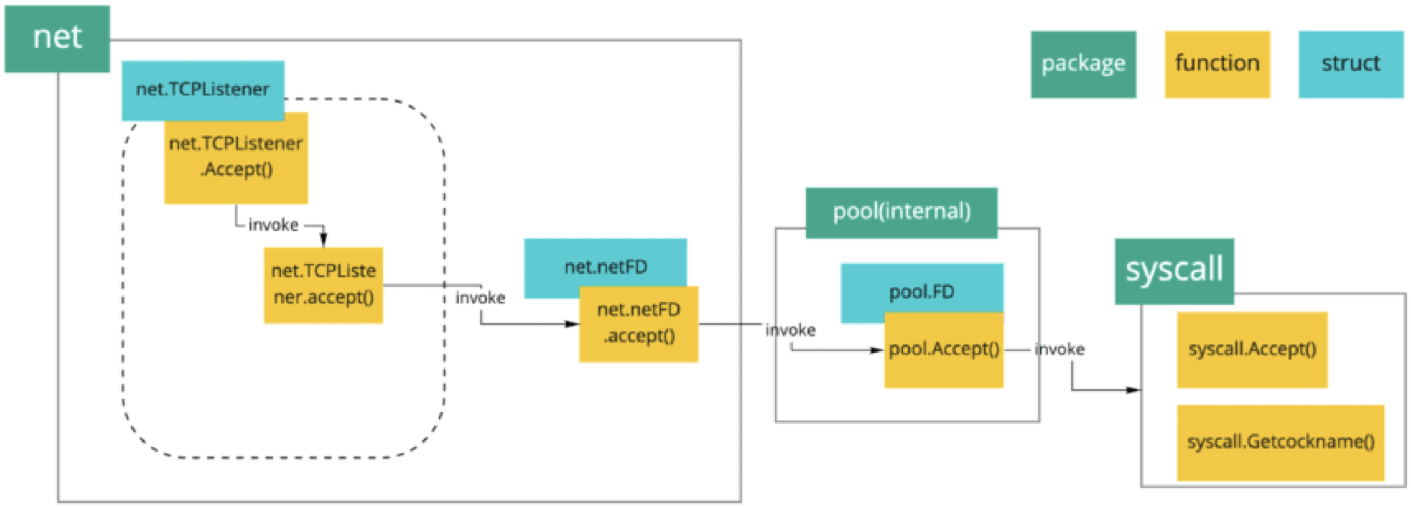
So far, you have understood the back side of starting to listen for TCP connections. Now, let's go back to the first sample code again.

```
func main() {
    addr := "localhost:8888"
    l, err := net.Listen("tcp", addr)

    // (omit)

    for {
        // Listen for an incoming connection
        conn, err := l.Accept()
        if err != nil {
            panic(err)
        }
        // Handle connections in a new goroutine
        // (omit)
    }
}
```

The next part is the mechanism to accept a connection, net.TCPListener.Accept.
net.TCPListener.Accept waits for the next call and returns a generic net.Conn. It implements the Accept method in the net.Listener interface that represents a generic network listener for stream-oriented protocols.

```
type Listener interface {
    Accept() (Conn, error)
    Close() error
    Addr() Addr
}
```

It is implemented by various types: FileListener, UnixListener and TCPListener.
In understanding this implementation, I have again prepared a map.



net.TCPListener.Accept invokes net.TCPListener.accept and then it invokes net.netFD.accept.

```
func (fd *netFD) accept() (netfd *netFD, err error) {
    d, rsa, errcall, err := fd.pfd.Accept()
    if err != nil {
        if errcall != "" {
            err = wrapSyscallError(errcall, err)
        }
        return nil, err
    }

    if netfd, err = newFD(d, fd.family, fd.sotype, fd.net); err != nil {
        poll.CloseFunc(d)
        return nil, err
    }
    if err = netfd.init(); err != nil {
        netfd.Close()
        return nil, err
    }
    lsa, _ := syscall.Getsockname(netfd.pfd.Sysfd)
    netfd.setAddr(netfd.addrFunc()(lsa), netfd.addrFunc()(rsa))
    return netfd, nil
}
```

Here, it executes two system calls: accept and getsockname.
The accept system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). As you see, SOCK_STREAM is chosen when it's a TCP listener. It extracts the first connection request on the queue of pending connections for listening socket, creates a new connected socket, and returns a new file descriptor referring to that socket.
The getsockname is called to know a new connected socket and file descriptor.

Read and write a connection

net.TCPListener.Accept returns net.Conn that is a generic stream-oriented network connection.

```
type Conn interface {
    Read(b []byte) (n int, err error)
    Write(b []byte) (n int, err error)
    Close() error
    LocalAddr() Addr
    RemoteAddr() Addr
    SetDeadline(t time.Time) error
    SetReadDeadline(t time.Time) error
    SetWriteDeadline(t time.Time) error
}
```

By using the net.Conn, you can read and write a connection like this:

```
func main() {
    addr := "localhost:8888"
    l, err := net.Listen("tcp", addr)

    // (omit)

    for {
        // Listen for an incoming connection
        conn, err := l.Accept()
        if err != nil {
            panic(err)
        }
        // Handle connections in a new goroutine
        go func(conn net.Conn) {
            buf := make([]byte, 1024)
            len, err := conn.Read(buf)
            if err != nil {
                fmt.Printf("Error reading: %#v\n", err)
                return
            }
            fmt.Printf("Message received: %s\n", string(buf[:len]))

            conn.Write([]byte("Message received.\n"))
            conn.Close()
        }(conn)
    }
}
```

Conclusion

List the key takeaways again.

With the net package, you can create a TCP server with such simple and little code (The Go playground).

If the host in the address parameter is empty or a literal unspecified IP address, listen on all available unicast and anycast IP addresses of the local system.

There are two kinds of DNS resolver: Go's built-in DNS resolver and uses Cgo DNS resolver. By default, Go's built-in DNS resolver is used.

If the port in the address parameter is empty or "0", a port number is automatically chosen by the system call bind.

Go accepts an incoming connection by using the system call accept.

You should now have a more in-depth understanding of the fundamentals of implementing a TCP server.