1. ## How do you handle panics and recover from them in Go?

   In the Go programming language, a panic is a run-time error that occurs when a program is unable to recover from an error. Panics are usually caused by programmer mistakes, such as trying to index an array out of bounds or trying to divide by zero.

   To handle a panic and recover from it, you can use the recover function inside a defer statement. The panic's error value can be retrieved by the recover function when a defer statement delays its execution until the surrounding function returns.

2. ## What is the "defer" keyword used for in Go?

   A staple in Golang interview questions for 2 years of experience, be prepared to answer this one. In the Go programming language, the defer keyword is used to defer the execution of a function until the surrounding function returns.

   The defer statement is used to ensure that a function is always executed, regardless of whether the surrounding function returns normally or through a panic. It is often used to perform cleanup tasks, such as closing a file or releasing a lock.

3. ## How do you create and use a package in Go?

   In the Go programming language, a package is a collection of related Go source files that are compiled together. Packages are used to organize and reuse code, and they provide a way to create and use libraries in Go.

   To create a package in Go, you simply put your Go source files in a directory with the same name as the package. The package name is the name of the directory in which the source files are located.

4. ## What is the difference between a package and a module in Go?

   In the Go programming language, a package is a collection of related Go source files that are compiled together. Packages are used to organize and reuse code, and they provide a way to create and use libraries in Go.

   A module is a unit of organization for Go source code, introduced in Go 1.11. Modules provide a way to manage dependency versions, and they are used to build, test, and publish Go packages.

   Modules are defined using a go.mod file, which specifies the module path, the module's dependencies, and the required versions of those dependencies.

5. How do you create a custom type in Go?

In the Go programming language, you can create a custom type by using the type keyword followed by the type name and the type you want to define it as.

Here is an example of creating a custom type called MyInt that is based on the built-in int type:

```
package main
type MyInt int
func main() {
  var x MyInt = 5
  fmt.Println(x)
}
```

This example creates a custom type called MyInt that is based on the int type. The custom type can be used like any other type in Go.

6. What is the syntax for type casting in Go?

The act of changing a value's type is known as "type casting" in the Go programming language. To cast a value to a different type in Go, you use the following syntax:

```
newType(expression)
```

Here is an example of type casting in Go:

```
package main
import "fmt"
func main() {
  var x float64 = 3.14
  var y int = int(x)
  fmt.Println(y)  // prints "3"
}
```

In this example, the value of x is cast from a float64 to an int. The result is the integer value 3.

7. How do you use the "blank identifier" in Go?

One of the most frequently posed Golang interview questions for experienced, be ready for it. The blank identifier is a unique identifier in the Go programming language that serves as a stand-in for when a value must be thrown away. The blank identifier is represented by an underscore character (_). It has no name and cannot be used as a variable.

Here is an example of using the blank identifier in Go:

```
package main
import "fmt"
func main() {
  // discard the error value
  _, err := os.Open("filename.txt")
  if err != nil {
 panic(err)
  }
}
```

In this example, the blank identifier is used to discard the value returned by os.Open, which is the file and an error. The error value is assigned to the err variable, while the file value is discarded using the blank identifier.

8. How do you create and use a pointer to a struct in Go?

Pointers in Go are variables that hold the memory address of other variables. Pointers are useful for passing variables by reference and for modifying variables through indirection. To create a pointer to a struct in Go, you use the & operator to get the memory address of the struct, and you use the * operator to define the type of the pointer:

```
package main

type Person struct {
  Name string
  Age  int
}
func main() {
  // create a pointer to a Person struct
  p := &Person{Name: "John", Age: 30}
  // modify the struct through the pointer
  p.Age = 31
  fmt.Println(p)  // prints "&{John 31}"
}
```

In this example, the p variable is a pointer to a Person struct. The struct is modified through the pointer by assigning a new value to the Age field.

9. How do you embed a struct in Go?

In the Go programming language, you can embed a struct inside another struct to create a composite data type. This is known as struct embedding.

To embed a struct, you simply specify the struct type as a field in the outer struct. The fields of the inner struct become fields of the outer struct, and the methods of the inner struct become methods of the outer struct.

10. How do you create and use a function closure in Go?

In Go, a function closure is a function that refers to variables from the scope in which it was defined. These variables are "closed over" by the function, and they remain accessible even after the function is invoked outside of their original scope.

Here's an example of how to create and use a function closure in Go:

```
package main
import "fmt"
func main() {
// Create a variable x and initialize it to 10
x := 10
// Create a function closure that captures the value of x
addX := func(y int) int {
   return x + y
}
// Use the function closure to add x to different values of y
fmt.Println(addX(5))  // prints 15
fmt.Println(addX(10)) // prints 20
// Modify the value of x and see that the function closure still uses the original
value
x = 20
fmt.Println(addX(5))  // still prints 15
}
```

11. What is the syntax for creating and using a function literal in Go?

An anonymous function is a function literal in Go, which is defined and called without a name. They can be assigned to a variable or passed as an argument to another function. Here's an example of how to create and use a function literal in Go:

```
package main
import "fmt"
func main() {
// create a function literal that takes two integers as input and returns their sum
add := func(x, y int) int {
   return x + y
}
// use the function literal to add two numbers
result := add(5, 10)
```

```
fmt.Println(result) // prints 15
// pass a function literal as an argument to another function
someFunc(func(x int) int {
   return x * x
})
}
func someFunc(f func(int) int) {
fmt.Println(f(5))
}
```

12. How do you use the "select" statement in Go?

In Go, the "select" statement is used to choose between multiple communication operations. It's similar to a "switch" statement, but it's used specifically for communication operations such as sending or receiving data on channels. The select statement blocks until one of its cases can run, then it runs that case, it's a blocking operation.

13. What is the syntax for creating and using a type assertion in Go?

In Go, a type assertion is used to check if an interface variable contains a specific type and, if it does, to extract the underlying value of that type.

Here's an example of how to create and use a type assertion in Go:

```
package main
import (
"fmt"
)

func main() {
// define an interface variable
var myvar interface{} = "hello"

// use type assertion to check the type of myvar
str, ok := myvar.(string)
if ok {
   fmt.Println(str)
} else {
   fmt.Println("myvar is not a string")
}
}
```

14. What is the syntax for creating and using a type switch in Go?

In Go, a type switch is used to check the type of an interface variable and to execute different code based on the type of the underlying value. A type switch is similar to a type assertion, but it can check for multiple types at once, and it doesn't require a variable to hold the underlying value.

Here's an example of how to create and use a type switch in Go:

```
package main
import (
"fmt"
)
func doSomething(i interface{}) {
switch v := i.(type) {
case int:
   fmt.Println("i is an int:", v)
case float64:
   fmt.Println("i is a float64:", v)
case string:
   fmt.Println("i is a string:", v)
default:
   fmt.Println("i is of an unknown type")
}
}
func main() {
doSomething(1)
doSomething(3.14)
doSomething("hello")
doSomething([]int{})
}
```

15. What is the syntax for creating and using a type conversion in Go?

One way to change the type of a value in Go is to apply a type conversion. Go has built-in type conversions for most of the basic types. Here is an example for how to create and use a type conversion in Go:

```
package main
import (
"fmt"
)
func main() {
var x float64 = 3.14
var y int = int(x) // type conversion from float64 to int
```

```
fmt.Println(y) // prints "3"
var a string = "42"
var b int64
b, _ = strconv.ParseInt(a, 10, 64) // type conversion from string to int64 using the standard library
fmt.Println(b) // prints "42"
}
```

16. How do you use the "sync" package to protect shared data in Go?

The "sync" package in Go provides various types and functions for synchronizing access to shared data. One of the most popular options is sync.Mutex, which ensures that only one goroutine at a time can access the shared data by means of mutual exclusion.

17. How do you use the "sync/atomic" package to perform atomic operations in Go?

The "sync/atomic" package in Go provides low-level atomic memory operations, such as atomic memory reads and writes, for use with the sync package. These operations allow you to perform atomic read-modify-write operations on variables that are shared across multiple goroutines, without the need for explicit locks or other synchronization.

18. How do you use the "context" package to carry around request-scoped values in Go?

The "context" package in Go provides a way to carry around request-scoped values and cancelation signals across API boundaries. It is often used in conjunction with http request handlers to provide request-scoped data such as request metadata, request timeout, etc.

19. How do you use the "net/http" package to build an HTTP server in Go?

The "net/http" package in Go provides a set of functions and types for building HTTP servers and clients. An easy HTTP server built with this software is demonstrated below.

```
package main
import (
"fmt"
"net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
fmt.Fprint(w, "Hello, World!")
}
func main() {
```

```
http.HandleFunc("/", handler)
http.ListenAndServe(":8080", nil)
}
```

This code creates an HTTP server that listens on port 8080 and handles incoming requests by calling the handler function.

20. How do you use the "encoding/json" package to parse and generate JSON in Go?

The encoding/json package in Go provides functionality for encoding and decoding JSON data.

To parse JSON data, you can use the json.Unmarshal() function. This function takes a byte slice of JSON data and a pointer to a struct, and it populates the struct with the data from the JSON.