98 Golang interview questions to hire talented interviewees

Questions

1. What are the key differences between `make` and `new` in Go?

2. How does Go handle concurrency, and what are goroutines and channels?

3. Explain the concept of interfaces in Go. Can you give an example?

4. What is the purpose of the `defer` keyword in Go?

5. How does Go manage memory, and what is garbage collection?

6. Describe the use of pointers in Go. When would you use them?

7. What are the different ways to declare variables in Go?

8. Explain the use of structs in Go. How are they different from classes in other languages?

9. What are the basic data types available in Go?

10. How do you handle errors in Go? What is the `error` type?

11. Explain the use of packages in Go. How do you import and use them?

12. What are the differences between arrays and slices in Go?

13. How does Go support testing? What is the `testing` package?

14. Explain the concept of methods in Go. How are they defined and used?

15. What is the zero value of a variable in Go? Does it differ by type?

16. How do you handle command-line arguments in Go?

17. Explain the purpose of the `go.mod` file in a Go project.

18. What is the use of the `select` statement in Go?

19. How do you implement a simple HTTP server in Go?

20. Explain the concept of closures in Go. Can you provide an example?

21. How does Go handle string manipulation? What are some common string functions?

22. What is the difference between a buffered and an unbuffered channel?

23. How can you detect race conditions in Go code?

24. Explain the difference between `make` and `new` in Golang. When would you use one over the other?

25. How does Golang handle concurrency? Describe goroutines and channels.

26. What is the purpose of the `select` statement in Golang? Provide a practical example.

27. Explain the concept of interfaces in Golang. How are they different from interfaces in other languages like Java or C#?

28. What are deferred functions in Golang? How do they work, and what are they typically used for?

29. Describe the use of context in Golang. How can it be used to manage goroutines?

30. Explain the purpose of the `go vet` tool. What kind of issues does it help identify?

31. What are the advantages of using Golang's built-in testing framework? How do you write a simple test case?

32. How do you handle errors in Golang? What is the purpose of the `error` interface?

33. Explain the concept of zero values in Golang. What is the zero value for different data types?

34. What is the role of the `init` function in Golang? When is it executed?

35. Describe how garbage collection works in Golang. Is it possible to manually trigger garbage collection?

36. How can you profile Golang code? What tools are available for performance analysis?

37. What are race conditions in concurrent Golang programs? How can you detect and prevent them?

38. Explain the use of mutexes in Golang. Provide an example of how to protect shared resources.

39. How does Golang support embedding? Give a practical use case.

40. What are the benefits of using static analysis tools in Golang? Give some examples of such tools.

41. Describe the purpose of the `go generate` command. How can it be used to automate code generation?

42. Explain the difference between buffered and unbuffered channels. When should you use each type?

43. What are the trade-offs between using shared memory and message passing for concurrency in Golang?

44. Describe how reflection works in Golang. What are its use cases and potential drawbacks?

45. Explain the purpose of the `go doc` tool. How can you document your Golang code effectively?

46. What is the purpose of the `iota` keyword in Golang? Provide an example of its use.

47. How can you implement a worker pool in Golang? What are the benefits of using a worker pool?

48. Explain the concept of method sets in Golang. How do they relate to interfaces?

49. Describe the process of building and deploying Golang applications. What are some common deployment strategies?

50. How does Golang's garbage collector work, and what are some strategies for minimizing garbage collection pauses in high-performance applications?

51. Explain the concept of 'escape analysis' in Go, and how it affects memory allocation and performance.

52. Describe the use cases for context.Context in Golang, and how it facilitates cancellation and deadline propagation across goroutines.

53. How does the Golang scheduler manage goroutines, and what are the implications for concurrency and parallelism?

54. What are the trade-offs between using channels and mutexes for synchronizing access to shared data in concurrent Go programs?

55. Explain the concept of 'zero-copy' networking in Golang, and how it can improve performance for network-intensive applications.

56. How can you implement a generic data structure (e.g., a generic stack or queue) in Go before the introduction of generics, and what are the limitations?

57. Discuss the differences between reflection and code generation in Golang, and when you might choose one over the other.

58. Explain how you would implement a custom linter in Golang to enforce specific coding standards or detect potential bugs.

59. Describe how you can use cgo to interface with C code in a Golang program, and what are the potential challenges?

60. How does Golang handle memory alignment, and why is it important for performance and portability?

61. Explain how you would debug a deadlock or race condition in a concurrent Golang program.

62. What are the benefits and drawbacks of using protocol buffers (protobufs) for data serialization in Golang?

63. Describe how you would implement a rate limiter in Golang to protect a service from being overwhelmed by requests.

64. How can you use the 'go:embed' directive to include static assets (e.g., HTML templates, images) in a Golang binary?

65. Explain how you would implement a custom error type in Golang that provides more context and debugging information.

66. Describe the use cases for the 'unsafe' package in Golang, and what are the risks associated with using it?

67. How does Golang's module system work, and how does it help manage dependencies in large projects?

68. Explain how you would implement a graceful shutdown mechanism for a Golang server application.

69. Describe how you can use the 'pprof' package to profile and optimize the performance of a Golang application.

70. How would you implement a worker pool pattern in Golang to manage and limit the number of concurrent goroutines?

71. Explain the concept of 'copy-on-write' semantics in Golang, and how it affects the behavior of slices and maps.

72. Describe how you would use build tags in Golang to conditionally compile code for different platforms or environments.

73. How can you implement a custom allocator in Golang to improve memory management for specific use cases?

74. How does Go's garbage collector work, and what strategies can you use to minimize its impact on performance-sensitive applications?

75. Explain the differences between `unsafe.Pointer`, `uintptr`, and `reflect.Value.UnsafeAddr()`. When should each be used, and what are the potential risks?

76. Describe the internal implementation of Go's maps. How do they handle collisions, and what are the performance implications of different key types?

77. How does Go's scheduler manage goroutines, and what factors can influence scheduling decisions?

78. Explain how Go's escape analysis works and how it impacts memory allocation and garbage collection.

79. Describe the role of the `runtime` package. Provide some examples of how its features can be used for advanced debugging or profiling.

80. What are the trade-offs between using channels and mutexes for synchronization in Go, and when would you choose one over the other?

81. How does Go's compiler optimize code, and what techniques can you use to help it generate more efficient binaries?

82. Explain how to use cgo effectively and what are the potential pitfalls of mixing Go and C code?

83. Describe the process of cross-compilation in Go and how to manage platform-specific dependencies.

84. How can you implement a custom memory allocator in Go and why might you want to do so?

85. Explain the concept of 'zero-copy' techniques in Go and how they can improve performance in I/O-bound applications.

86. How does Go support dynamic linking, and what are the advantages and disadvantages of using it?

87. Describe the differences between reflection and code generation in Go, and when would you choose one approach over the other?

88. Explain how Go's race detector works, and how can you use it effectively to identify concurrency issues?

89. What are the limitations of Go's type system, and how can generics (if available) address some of those limitations?

90. How does Go handle signals, and how can you use them to gracefully shut down a program?

91. Explain the concept of 'context' in Go and how it is used for request cancellation and deadline propagation.

92. How can you implement a custom linter for Go code and why might you want to do so?

93. Describe the different ways to profile Go code and how to interpret the profiling data.

94. Explain how Go's error handling mechanism works, and what are the best practices for handling errors in production code?

95. How do you implement a worker pool in Go, and what are the key considerations for designing an efficient worker pool?

96. Explain the differences between blocking and non-blocking I/O in Go, and when would you use each approach?