1. Какой самый эффективный способ конкатенации строк?

1) Использование функции fmt.Sprintf

| str1 := "Hello, " str2 := "World!" result := fmt.Sprintf("%s%s", str1, str2) |
| --- |

2) Использование strings.Builder

| var builder strings.Builder  builder.WriteString("Hello, ")  builder.WriteString("World!")  result := builder.String() |
| --- |

2. Что такое интерфейсы, как они применяются в Go?

Интерфейс в Go - это контракт, определяющий набор методов, которые должны быть реализованы типами, чтобы соответствовать этому интерфейсу. Являются инструментами достижения полиморфизма.

Основные характеристики интерфейсов в Go:

* Определение контракта: интерфейс определяет контракт, который должен быть реализован любым типом, желающим соответствовать этому интерфейсу. Контракт состоит из набора методов, которые должны быть реализованы.
* Неявная реализация: любой тип, который реализует все методы, определенные в интерфейсе, автоматически и неявно реализует этот интерфейс. Нет необходимости явно заявлять, что тип реализует интерфейс.
* Универсальность: интерфейсы позволяют создавать универсальный код, который может работать с любым типом, соответствующим заданному интерфейсу, не зная его конкретной реализации.
* Динамическая типизация: интерфейсы в Go обеспечивают динамическую типизацию. Переменная интерфейсного типа может хранить значение любого типа, который реализует этот интерфейс.

3. Чем отличаются RWMutex от Mutex?

RWMutex (блокировка чтения-записи) и Mutex (взаимное исключение) в Go отличаются поведением и предназначением.

1) Mutex:

- Mutex предназначен для защиты единственного ресурса от одновременного доступа.

- Mutex позволяет любой горутине войти в критическую секцию, но только одной горутине за раз.

- При попытке входа в критическую секцию, занятую другой горутиной, текущая горутина будет заблокирована до тех пор, пока mutex не будет освобожден.

2) RWMutex:

- RWMutex предназначен для защиты ресурса, в котором одновременно осуществляется чтение и запись.

- RWMutex позволяет нескольким горутинам одновременно читать данные, но только одной горутине за раз писать данные.

- При попытке записи в ресурс, занятый другой горутиной для записи, текущая горутина будет заблокирована до тех пор, пока RWMutex не будет освобожден.

- При попытке записи в ресурс, занятый другими горутинами для чтения, текущая горутина будет заблокирована до тех пор, пока все читатели не освободят RWMutex.

Основные различия:

- Параллелизм: Mutex позволяет только одной горутине за раз получить доступ к ресурсу, RWMutex позволяет нескольким горутинам одновременно читать данные, но только одной горутине за раз писать данные.

- Производительность: RWMutex более производителен, чем Mutex, когда большинство доступов к ресурсу являются операциями чтения. Целесообразно использовать RWMutex при работе с map.

4. Чем отличаются буферизированные и не буферизированные каналы?

Не буферизированные каналы - это каналы, которые не имеют внутреннего буфера и требуют блокировки отправителя или получателя при передаче данных. Отправитель будет заблокирован до тех пор, пока получатель не прочитает данные из канала, и наоборот - получатель будет заблокирован до тех пор, пока отправитель не передаст данные.

В небуферизованном канале посылающий может определить факт получения сообщения.

Для создания не буферизированного канала в Go используется следующий синтаксис:

ch := make(chan int) (int взят для примера, может использоваться любой тип данных)

Буферизированные каналы

Буферизированные каналы - это каналы, которые имеют внутренний буфер и позволяют отправителю передавать данные, даже если получатель еще не готов к их приему. Буферизированные каналы в Go позволяют уменьшить количество блокировок и увеличить производительность программы.

Для создания буферизированного канала в Go используется следующий синтаксис:

ch := make(chan int, 10) (int и 10 взяты для примера, может быть любой типа данных и размер канала)

5. Какой размер у структуры struct{}{}?

Размер структуры struct{}{} - 0 байт. Оправданно ее применение в случаях, когда требуется минимальное потребление памяти. Например:

1. представление элементов множества;
2. сигнализация о наличии чего-либо, без необходимости передачи дополнительной информации;

6. Есть ли в Go перегрузка методов или операторов?

В Go нет прямой поддержки перегрузки методов или операторов.

При использовании языка Go рекомендуется использовать несколько методов с разными названиями для достижения того же эффекта.

Возможна перегрузка методов родителя методами потомка.



| // родитель type A struct {}  // может быть переопределен в потомке func (a \*A) CallFirst() {  fmt.Println("A CallFirst") } // потомок type B struct {  A }  // переопределяем метод в потомке func (b \*B) CallFirst() {  fmt.Println("B CallFirst") }  a := new(A) a.CallFirst() // "A CallFirst" b := new(B) b.CallFirst() // "B CallFirst" |
| --- |

7. В какой последовательности будут выведены элементы map[int]int?  
  
 *Пример:*

| m[0]=1 m[1]=124 m[2]=281 |
| --- |

Ответ: в случайной, т.к. map - неупорядоченная структура данных и не сохраняет порядок добавления элементов.

8. В чем разница make и new?

make и new являются двумя разными встроенными функциями, используемыми для выделения памяти под различные типы данных. Основная разница между ними заключается в том, что они используются для разных типов:

1. make:

- Функция make используется для выделения памяти под встроенные изменяемые типы данных: slice, map и channel.

- make инициализирует эти типы данных и возвращает созданный объект.

- Например:

| s := make([]int, 0, 10) |
| --- |

создает новый срез (slice) с начальной емкостью 10.

1. new:

- Функция new используется для выделения памяти под встроенные неизменяемые типы данных, такие как int, float, bool, string и другие.

- new выделяет память для нового экземпляра указанного типа, инициализирует его нулевым значением и возвращает указатель на этот экземпляр.

- Например: p := new(int) создает новую переменную типа int и возвращает указатель на нее.

Основные различия:

1. Типы данных: make используется для изменяемых типов данных, new - для неизменяемых.

2. Инициализация: make инициализирует тип данных, new возвращает указатель на нулевое значение типа.

3. Возвращаемое значение: make возвращает сам объект, new возвращает указатель на объект.

9. Сколько существует способов задать переменную типа slice или map?

Существует два способа:

1. При помощи функции make:

Например, создаем новый слайс длиной 0 и емкостью (capacity) 10:

| make([]int, 0, 10) |
| --- |

Или создаем новую мапу:  
m := make(map[string]bool, 0) с начальной емкостью 0.

1. Использование составных литералов:

| s := []int{1, 2, 3}  m := map[string]bool{ "a" : true, "b" : false, } |
| --- |

В фигурных скобках возможно указывать элементы, входящие в состав слайса или мапы.

1. Пустое объявление:

Для слайсов: var slice []type

Для мап: var map map[keytype]valuetype

1. Нулевые литералы:

Для слайсов: var slice []type = nil

Для мап: var map map[keytype]valuetype = nil

10. Что выведет данная программа и почему?

| func update(p \*int) {  b := 2  p = &b }  func main() {  var (  a = 1  p = &a  )  fmt.Println(\*p)  update(p)  fmt.Println(\*p) } |
| --- |

Ответ:

1

1

1. В функции main() объявляется переменная a со значением 1 и создается указатель p, который указывает на a. Первый fmt.Println(\*p) выводит значение, на которое указывает p, то есть 1.

2) В функции update(p \*int):

- Объявляется локальная переменная b со значением 2 и создается указатель, который указывает на b.

- Оператор p = &b изменяет локальную копию указателя p, полученную в качестве аргумента. Но это не влияет на оригинальный указатель p в main(), т.к. в Go аргументы передаются по значению.

3. После вызова update(p) в main():

- Второй fmt.Println(\*p) все еще выводит 1, потому что оригинальный указатель p в main() не был изменен вызовом update(p).

Изменение локальной копии указателя p в update(p \*int) не влияет на оригинальный указатель p в main().

11. Что выведет данная программа и почему?

| func main() {  wg := sync.WaitGroup{}  for i := 0; i < 5; i++ {  wg.Add(1)  go func(wg sync.WaitGroup, i int) {  fmt.Println(i)  wg.Done()  }(wg, i)  }  wg.Wait()  fmt.Println("exit") } |
| --- |

Вывод программы будет непредсказуемым, но наиболее вероятный вывод будет следующим:

0

1

2

3

4

exit

В цикле for создается 5 горутин, каждая из которых выполняет анонимную функцию.

Внутри анонимной функции значение переменной i передается как аргумент, чтобы каждая горутина имела свою копию i.

Проблема в том, что sync.WaitGroup также передается как аргумент в анонимную функцию по значению, а не по ссылке. Это значит, что каждая горутина работает со своей локальной копией wg, а не с оригинальным wg из main().

В результате, когда горутины вызывают wg.Done(), они уменьшают счетчик своей локальной копии wg, а не оригинального wg из main(). Поэтому wg.Wait() в main() никогда не завершится.

Горутины успевают выполнить fmt.Println(i) до того, как программа завершится. Поэтому вывод будет содержать числа от 0 до 4, а затем "exit".

Чтобы исправить эту проблему, sync.WaitGroup должна быть передана по ссылке, а не по значению:

| func main() {  wg := &sync.WaitGroup{}  for i := 0; i < 5; i++ {  wg.Add(1)  go func(wg \*sync.WaitGroup, i int) {  fmt.Println(i)  wg.Done()  }(wg, i)  }  wg.Wait()  fmt.Println("exit") } |
| --- |

В этом случае все горутины будут работать с одним и тем же wg, и wg.Wait() в main() будет ожидать, пока все горутины не завершатся, прежде чем вывести "exit".

12. Что выведет данная программа и почему?

| func main() {  n := 0  if true {  n := 1  n++  }  fmt.Println(n) } |
| --- |

Ответ: 0, т.к. в блоке с условием (if) создается новая переменная n с областью видимости только внутри данного блока if. Оригинальная n остается неизменной и равной 0. После выхода из if блока, внутренняя переменная n с областью видимости внутри if блока перестает существовать.

Поэтому при вызове fmt.Println(n) будет выведено значение 0, которое было изначально присвоено переменной n в main() функции.

Это связано с тем, что в Go переменные, объявленные внутри блоков, if, for, switch и т.д., имеют локальную область видимости и не затрагивают переменные с тем же именем, объявленные на более высоком уровне.

13. Что выведет данная программа и почему?

| func someAction(v []int8, b int8) {  v[0] = 100  v = append(v, b) }  func main() {  var a = []int8{1, 2, 3, 4, 5}  someAction(a, 6)  fmt.Println(a) } |
| --- |

Программа выведет:

[100 2 3 4 5]

В функции someAction(v []int8, b int8): Первый элемент среза a изменяется на 100.

Новый элемент b (со значением 6) добавляется к срезу с помощью append(v, b). Однако изначально размер и емкость среза а были неявно инициализированы как 5; при добавлении нового элемента в слайс был инициализирован новый массив (новый указатель на расположение в памяти, len = 6, capacity x2 от предыдущего слайса - 10). Таким образом, в первоначальном слайсе а изменится только начальный элемент.

14. Что выведет данная программа и почему?

| func main() {  slice := []string{"a", "a"}   func(slice []string) {  slice = append(slice, "a")  slice[0] = "b"  slice[1] = "b"  fmt.Print(slice)  }(slice)  fmt.Print(slice) } |
| --- |

В функции создаем новый слайс с именем slice при выполнении операции append, т.к. у первоначального слайса len = 2 и capacity =2, следовательно, для добавления нового элемента требуется создавать новый массив с capacity 4. После это меняем первые два элемента нового слайса и выводим его в stdout:

{"b", "b", “a”}

После этого выводим оригинальный неизмененный слайс из main:

{"a", "a"}