Golang WB L1 практика

1. Какой самый эффективный способ конкатенации строк?

`strings.Builder` – это более эффективный метод конкатенации строк, особенно при работе с большим объемом данных. Он минимизирует количество выделений памяти, что делает его предпочтительным выбором для высокопроизводительных операций.

Для запуска этих бенчмарков, нужно использовать команду `go test -bench=.`

```
package main

import (
    "strings"
    "testing"
)

func BenchmarkStringBuilder(b *testing.B) {
    str1 := "Hello, "
    str2 := "World!"

    for i := 0; i < b.N; i++ {
        var sb strings.Builder
        sb.WriteString(str1)
        sb.WriteString(str2)
        _ = sb.String()
    }
}
```

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

Интерфейс представляет собой абстракцию поведения других типов, они определяют функционал, но не реализуют его. Например, у нас есть объект,

содержащий множество подгрупп. Этот объект определяет общий функционал для них, но сама реализация функционала будет зависеть от конкретных подгрупп.

```
type Vehicle interface{
    move()
}

// структура "Автомобиль"
type Car struct{ }

// структура "Самолет"
type Aircraft struct{}


func (c Car) move(){
    fmt.Println("Автомобиль едет")
}
func (a Aircraft) move(){
    fmt.Println("Самолет летит")
}

func main() {

    var tesla Vehicle = Car{}
    var boing Vehicle = Aircraft{}
    tesla.move()
    boing.move()
}
ИЛИ ЖЕ
Интерфейс, который не содержит ни одного
метода называется пустым интерфейсом: interface{}.
 Пустой интерфейс может содержать значение любого типа.
 Пустые интерфейсы используются в коде, где необходимо работать
 значениями неизвестного типа
 var i interface{} = 12
```

```
if v, ok := i.(int); ok {
    fmt.Println(v+12) // Суммирование не произойдет, если ok ==
}
```

3. Чем отличаются RWMutex от Mutex?

RWMutex нужен, когда у нас есть объект, который нельзя параллельно писать, но можно параллельно читать. Например, стандартный тип map.

- `sync.Mutex` - блокирует кусок кода как на запись, так и на чтение;
- `sync.RWMutex` - позволяет блокировать кусок кода только на запись.

4. Чем отличаются буферизированные и не буферизированные каналы?
    a. **Не буферизированные каналы**:
        - **Передача данных**: Передача данных происходит непосредственно от отправителя к получателю. Отправляющая горутина блокируется до тех пор, пока другая горутина не получит данные из канала. Аналогично, получающая горутина блокируется до тех пор, пока данные не будут отправлены в канал.
        - **Синхронизация**: Они обеспечивают строгую синхронизацию между отправляющей и получающей горутинами.
        - **Использование**: Хорошо подходят для ситуаций, где важна синхронизация и непосредственная передача данных.
    b. **Буферизированные каналы**:
        - **Передача данных**: Передача данных осуществляется через буфер, что позволяет отправляющей горутине продолжать выполнение без ожидания получения данных другой горутиной, если в буфере есть свободное место. Получающая горутина блокируется только тогда, когда буфер пуст, а отправляющая горутина блокируется, когда буфер полон.
        - **Асинхронность**: Они позволяют некоторую степень асинхронности между отправкой и получением данных, поскольку

данные могут храниться в буфере.

- **Размер буфера**: Размер буфера задается при создании канала, например, `make(chan int, 10)` создает буферизированный канал для 10 целых чисел.
- **Использование**: Хорошо подходят для ситуаций, где требуется асинхронная передача данных или сглаживание разницы в скорости обработки между отправляющей и получающей горутинами.

Примеры:

- **Не буферизированный канал**:

```
goКопировать код
ch := make(chan int)
```

- **Буферизированный канал**:

```
goКопировать код
ch := make(chan int, 5)
```

В зависимости от задач и требований вашей программы вы можете выбрать подходящий тип канала.

5. Какой размер у структуры struct{}{}?
   Размер структуры `struct{}{}` в Go равен нулю байт. Это так называемая пустая структура. Она не занимает места в памяти, поскольку не содержит никаких полей. Пустые структуры часто используются в Go для обозначения наличия чего-либо без хранения данных, например, в каналах или в качестве значений в картах (map).

```
package main

import (
    "fmt"
    "unsafe"
)

type S1 struct {
    f1 int
}

func main() {
    s1 := S1{}
    s2 := struct{}{}

    fmt.Printf("s1 size: %v\n", unsafe.Sizeof(s1))
    fmt.Printf("s2 size: %v\n", unsafe.Sizeof(s2))
}
```

6. Есть ли в Go перегрузка методов или операторов?

В Go нет перегрузки методов или операторов. Это означает, что вы не можете создавать несколько функций или методов с одинаковыми именами, но разными типами параметров или количеством параметров, как это возможно в некоторых других языках программирования, таких как C++ или Java. В Go каждая функция или метод должно иметь уникальное имя.

Для достижения аналогичного функционала обычно используют разные подходы, такие как создание функций с разными именами, использование интерфейсов или применение вариативных параметров.

1. В какой последовательности будут выведены элементы map[int]int?

   Пример:

m[0]=1

m[1]=124

m[2]=281

Так как map - это неупорядоченная коллекция. то значения будут выдаваться рандомно

```
package main

import (
    "fmt"
)

func main() {
    m := make(map[int]int)

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

    // Итерация по элементам map
    for key, value := range m {
        fmt.Printf("Ключ: %d, Значение: %d\n", key, value)
    }
}
```

8. В чем разница make и new?

Make и new – это встроенные механизмы для выделения памяти, каждый из которых используется в разных ситуациях и имеет свои особенности.
new инициализирует нулевое значение для данного типа и возвращает указатель на этот тип.
С другой стороны, make используется исключительно для создания и инициализации срезов, отображений и каналов. Он возвращает инициализированный экземпляр указанного типа, готовый к использованию после создания.
Основное различие между ними заключается в том, что make возвращает готовый к использованию инициализированный тип, в то время как new возвращает указатель на тип с его нулевым значением.

a := new(chan int) // a имеет тип *chan int
b := make(chan int) // b имеет тип chan int

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

Способы задать переменную типа `slice`

1. **Инициализация пустого слайса**:

```
var s []int
```

2. **Инициализация слайса с определенной длиной и нулевыми значениями**:

```
s := make([]int, 5)  // Создает слайс длиной 5
```

3. **Инициализация слайса с определенной длиной и емкостью**:

```
s := make([]int, 5, 10)  // Создает слайс длиной 5 и емкос
тью 10
```

4. **Литеральная инициализация слайса с известными значениями**:

```
s := []int{1, 2, 3, 4, 5}
```

5. **Инициализация слайса из существующего массива**:

```
a := [5]int{1, 2, 3, 4, 5}
s := a[:]
```

Способы задать переменную типа `map`

1. **Инициализация пустой карты**:

```
var m map[string]int
```

2. **Инициализация карты с использованием функции `make`**:

```
m := make(map[string]int)
```

3. **Литеральная инициализация карты с известными значениями**:

```
m := map[string]int{"one": 1, "two": 2, "three": 3}
```

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)
}
```

Функция вернёт два значения: 1. Переменная p содержит указатель на переменную a.
2. В функции update создаётся переменная b, указатель на которую будет передан в переменную p.
Однако после завершения работы функции update переменная b и указатель на неё будут удалены.

11. Что выведет данная программа и почему?

Будет возникать ошибка при передаче WaitGroup в анонимную функцию. Вместо того, чтобы передавать указатель на WaitGroup, программа передает её копию. Это приводит к тому, что каждая горутина работает со своей

копией WaitGroup, что делает ожидание в wg.Wait() бесполезным, поскольку оригинальная WaitGroup в main() никогда не завершится.

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

12. Что выведет данная программа и почему?

Внутри блока `if` будет создана еще одна переменная `n`, которая будет иметь локальную область видимости и будет существовать только в пределах этого блока. Поэтому результат будет 0.

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

13. Что выведет данная программа и почему?

Если прочитать определение `append`, становится ясно, что при выходе за пределы емкости среза создается новый срез на основе старого с добавлением нового элемента. Однако в данном случае ничего не изменится, так как `v` внутри функции `someAction` является копией ссылки на срез `a`, и любые изменения, внесенные в `v`, не влияют на переменную `a` в функции `main`.

```
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.Prin
```

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)
}
```

Когда функция-замыкание вызывается со срезом `slice`, это создает копию ссылки на оригинальный срез. Поэтому изменения, внесенные в `slice` внутри функции-замыкания, не будут отражаться на оригинальном срезе вне функции.