Урок Спринт 2.15

Atomic

Атомики в Go — это механизмы, которые позволяют выполнять операции над примитивными типами данных без блокировок и синхронизации. Они обеспечивают атомарную и непрерывную работу с разделяемыми данными, что делает их полезными для реализации `lock-free` структур данных.

Пакет `sync/atomic` в Go предоставляет функции для выполнения атомарных операций, таких как чтение, запись, добавление и сравнение-замена, над примитивными типами данных, такими как целочисленные и указатели. Эти функции гарантируют, что операции выполняются непрерывно и не могут быть прерваны другими горутинами.

Преимущества использования атомиков в Go:

- Избегание блокировок и мьютексов, что может привести к повышению производительности и уменьшению накладных расходов.
- Обеспечение безопасности при работе с разделяемыми данными, так как атомики предоставляют гарантии атомарности операций.
- Поддержка параллельного выполнения без необходимости в явной синхронизации.

Пример использования пакета `sync/atomic`:

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

var counter int32
var wg sync.WaitGroup

func main() {
    // Увеличиваем значение counter с помощью атомарной операции AddInt32
    wg.Add(2)
    go increment()
```

```
    go increment()
    wg.Wait()
    fmt.Println("Counter:", counter)
}

func increment() {
    defer wg.Done()
    atomic.AddInt32(&counter, 1)
}
```

В этом примере мы создаём горутины, которые одновременно увеличивают значение переменной `counter` с помощью атомарной операции `AddInt32`. Функция `increment` вызывается из двух горутин параллельно, и благодаря использованию атомиков, мы можем безопасно и эффективно выполнять операции над разделяемыми данными.

Операция `LoadPointer` в пакете `sync/atomic` используется для атомарной загрузки указателя из памяти. Эта операция гарантирует, что значение указателя будет прочитано непрерывно и не будет изменено другими горутинами во время чтения.

Пример использования `LoadPointer`:

```
package main

import (
    "fmt"
    "sync/atomic"
    "unsafe"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    person := &Person{Name: "Alice", Age: 25}

    // Загружаем указатель на структуру Person
    ptr := unsafe.Pointer(person)
    loadedPtr := atomic.LoadPointer(&ptr)

    // Преобразуем указатель обратно в структуру Person
    loadedPerson := (*Person)(loadedPtr)

    fmt.Println(loadedPerson.Name, loadedPerson.Age)
}
```

В этом примере мы создаем структуру `Person` и сохраняем её указатель в переменной `person`. Затем мы загружаем указатель с помощью `atomic.LoadPointer(&ptr)`. Загружённый указатель преобразуется обратно в структуру `Person`, чтобы получить доступ к её полям.

Функция `CompareAndSwapPointer` используется для атомарного сравнения и замены указателя. Она сравнивает значение указателя с ожидаемым значением и, если они совпадают, заменяет его на новое значение. Эта операция обычно используется для реализации примитивов синхронизации.

Пример использования `CompareAndSwapPointer`:

```
package main

import (
    "fmt"
    "sync/atomic"
    "unsafe"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    person := &Person{Name: "Alice", Age: 25}

    // Загружаем указатель на структуру Person
    ptr := unsafe.Pointer(person)

    // Ожидаемое значение указателя
    expectedPtr := ptr

    // Новое значение указателя
    newPtr := unsafe.Pointer(&Person{Name: "Bob", Age: 30})

    // Атомарно сравниваем и заменяем указатель
    swapped := atomic.CompareAndSwapPointer(&ptr, expectedPtr, newPtr)

    if swapped {
        fmt.Println("Pointer was successfully swapped")
    } else {
        fmt.Println("Pointer was not swapped")
    }
}
```

Создаём структуру `Person` и сохраняем её указатель в переменной `person`. Далее мы загружаем указатель в переменную `ptr` и указываем ожидаемое значение `expectedPtr`. Создаём новый указатель `newPtr` на структуру `Person` с другими значениями полей.

Затем мы вызываем `atomic.CompareAndSwapPointer(&ptr, expectedPtr, newPtr)` для атомарного сравнения и замены указателя. Если значение указателя `ptr` совпадает с ожидаемым значением `expectedPtr`, оно заменяется на новое значение `newPtr`. Результат операции `CompareAndSwapPointer` сохраняется в переменной `swapped`, которая позволяет нам узнать, была ли успешно выполнена замена указателя.

Справка