GO-10 04: Пакет unsafe

Описание:

Мы с вами узнали, что Go является типизированным языком, в котором реализованы указатели. Наряду с многопоточностью, асинхронностью и богатым функционалом для тестирования, это позволяет программистам создавать безопасные и производительные программы, не заостряя свое внимание, например, на том, где именно будет выполняться программа.
Также вы помните, что, несмотря на наличие указателей, Go не поддерживает адресную арифметику. Но на самом деле, не все так просто и есть возможность обойти некоторые из этих ограничений. Для этого надо спуститься на более низкий уровень и прежде, чем это сделать, узнать о возможностях и предназначении интересного и необычного пакета unsafe.
Необычность пакета unsafe заключается в том, что вы не найдете код реализации его методов в стандартной поставке языка. Перейдя по ссылке https://golang.org/src/unsafe/unsafe.go, можно увидеть лишь названия и описание методов. Это обусловлено тем, что функционал пакета unsafe реализуется компилятором.
Возможности пакета unsafe позволяют получить доступ непосредственно к памяти, тем самым обойти строгие ограничения инструментов контроля типов языка. Но именно это накладывает ряд ограничений на программы, которые импортируют пакет unsafe.
Какие это могут быть ограничения?
Для начала, согласно опубликованным рекомендациям совместимости программ на Go для версии 1 (https://golang.org/doc/go1compat#expectations) не гарантируется выполнение кода, использующего пакет unsafe, при изменении реализации в новых версиях языка. То есть разработчики языка прямо заявляют, что в процессе разработки новых версий языка могут быть внесены изменения, которые не являются обратно совместимыми, что может повлечь за собой поломку уже написанного функционала.
Кроме этого, ваша программа может стать платформозависимой, потому что результат работы вашего кода, с использованием пакета unsafe, может зависеть от архитектуры платформы.
Само название пакета говорит о том, что его использовать небезопасно и всю ответственность за его применение программист полностью возлагает на себя. Например, из-за неправильно построенного алгоритма можно совершенно случайно получить доступ к произвольной области памяти, в которой на момент выполнения кода могут находиться произвольные данные, в том числе и конфиденциальные.
Именно из-за этого факта пакет unsafe может быть заблокирован у облачных провайдеров, что сделает невозможным разворачивание вашего сервиса на мощностях такого провайдера.
И наконец — баги. В любой программе случаются ошибки и современные высокоуровневые языки программирования и инструменты отладки всячески помогают нам в их поиске и устранении. Но в случае использования пакета unsafe все эти средства

могут оказаться бесполезными. Все потому что ошибки, возникающие при работе с памятью, зачастую бывают трудновоспроизводимыми, так как зависят от многих факторов.
Все это является платой за возможность оптимизации наиболее проблемных и узких мест вашей программы. Применять пакет unsafe нужно в том случае, когда ваша программа уже оптимизирована, когда вы достигли определенных ограничений и знаете, для чего вам нужно применить пакет unsafe. И даже в этом случае его применение должно ограничиваться минимально возможной областью кода.
Основные функции пакета
Назначение пакета unsafe в том, чтобы предоставить вам инструмент для доступа к памяти. Для этого в пакете имеется всего несколько функций.
unsafe.Sizeof()
Функция Sizeof() принимает в качестве аргумента значение любого типа. Результатом ее работы является размер памяти в байтах, который выделяется под переменную заданного типа. Он не включает ту память, на которую может ссылаться переданный аргумент.
Посмотрим на пример.

```
str1 := "some text"
str2 := "more some text"

fmt.Println("Sizeof str1:", unsafe.Sizeof(str1))
fmt.Println("Sizeof str2:", unsafe.Sizeof(str2))
fmt.Println("Len str1:", len(str1))
fmt.Println("Len str2:", len(str2))

sl1 := []int16{1, 2, 3}
sl2 := []int16{4, 5, 6, 7, 12}

fmt.Println("Sizeof sl1:", unsafe.Sizeof(sl1))
fmt.Println("Sizeof sl2:", unsafe.Sizeof(sl2))
fmt.Println("Len sl1:", len(sl1))
fmt.Println("Len sl2:", len(sl2))
```

Если выполнить этот код, то мы увидим довольно интересный вывод.

```
Sizeof str1: 16
Sizeof str2: 16
Len str1: 9
Len str2: 14
Sizeof sl1: 24
Sizeof sl2: 24
Len sl1: 3
Len sl2: 5
```

То есть Sizeof() возвращает размер фиксированной части типа переданного аргумента, в которую не входят, например, содержимое строки или массив, на который ссылается слайс.
Результат работы Sizeof() является константой и в некоторых случая может быть использован для вычисления, например, размерности массива при считывании данных структуры из памяти.
Посмотрим на небольшой пример.

```
package main

import (
    "fmt"
    "unsafe"
)

type St struct {
    a int64
    f float64
    c int16
}

const size = unsafe.Sizeof(St{})

func main() {
    st := St{
        5,
        0.1,
        1,
    }
    bytes := (*[size]byte)(unsafe.Pointer(&st))
    st1 := (*St)(unsafe.Pointer(&bytes[0]))

    fmt.Printf("Origin struct: %#v\n", st)
    fmt.Printf("Bytes: %#v\n", bytes)
    fmt.Printf("Struct: %#v\n", *st1)
}
```

Запустив программу, мы увидим следующий вывод.

```
Origin struct: main.St{a:5, f:0.1, c:1}
```

```
Bytes: &[24]uint8{0x5, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x9a, 0x99,
0x99, 0x99, 0x99, 0x99, 0xb9, 0x3f, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0}
Struct: main.St{a:5, f:0.1, c:1}
```

В примере видно, что в начале программы мы применили Sizeof() для определения размера гипотетической переменной этого типа и присвоили это значение константе size. Далее инициализировали перменную st нашей структурой с некоторым набором полей. А затем произвели некоторые интересные манипуляции. Получили указатель на структуру при помощи unsafe.Pointer и далее при помощи операции преобразования типов считали нужное количество байт. Результатом этого выражения стал указатель на массив байт. Следующим шагом мы получили указатель на элемент с индексом 0 и при помощи все той же операции преобразования типов преобразовали последовательность байт в структуру типа St, взяли указатель на нее и присвоили его в переменную st1.
Об указателе unsafe.Pointer подробнее узнаем немного позже.
unsafe.Alignof()
Теперь поговорим немного о представлении данных и их выравнивании в памяти. Память — это некая байтовая последовательность с адресным доступом. Данные хранятся в памяти в виде последовательности байт. Соответственно, чтобы считать какие-то данные из памяти, необходимо обратиться к ним по адресу. Но устройство компьютера позволяет считать за раз какой-то фиксированный размер данных. Это обуславливается архитектурой процессора. Современные 64-битные компьютеры могут считывать за раз 8 байт информации, а предыдущее поколение 32-битных компьютеров — только 4 байта. Этот фиксированный размер считываемой информации называют машинным словом. Таким образом, логично предположить, что для более эффективной работы данные в памяти должны располагаться соразмерно машинному слову, то есть быть выровнены таким образом, чтобы значение многобайтового типа не пересекло границы машинного слова.
Цель выравнивания — расположить последовательность байт данных разных типов составных структур так, чтобы доступ к ним был более оптимальный. Обратной стороной выравнивания является неоптимальное использование памяти, так как для выравнивания используются нули.
Выравнивание структур выполняется по самому большому полю. Зная то, что выравнивание обеспечивается заполнением памяти нулями, легко предположить, что в некоторых случаях количество занятой памяти может сильно превышать размер самих данных в этой структуре.
unsafe.Offsetof()
Функция Offsetof() для поля структуры возвращает смещение поля относительно начала структуры, которая его содержит.
На примере структуры Customer, которая нам уже встречалась в предыдущих разделах, рассмотрим следующий код.

```
package main
```

```
import (
    "fmt"
    "unsafe"
)

type Customer struct {
    Name string
    Age int16
    Balance int
    Discount bool
    Debt int
}


func main() {
    cust := Customer{
        Name:         "New Customer",
        Age:          25,
        Balance:      15000,
        Debt:         3000,
        Discount:     true,
    }

    fmt.Printf("Bytes %#v\n",
(*[unsafe.Sizeof(Customer{})]byte)(unsafe.Pointer(&cust)))

    fmt.Printf("Name: Offsetof: %d Sizeof %d Alignof: %d\n",
unsafe.Offsetof(cust.Name), unsafe.Sizeof(cust.Name),
unsafe.Alignof(cust.Name))
    fmt.Printf("Age: Offsetof: %d SizeOf: %d, Alignof: %d\n",
unsafe.Offsetof(cust.Age), unsafe.Sizeof(cust.Age),
unsafe.Alignof(cust.Age))
    fmt.Printf("Balance: Offsetof: %d SizeOf: %d, Alignof: %d\n",
unsafe.Offsetof(cust.Balance), unsafe.Sizeof(cust.Balance),
unsafe.Alignof(cust.Balance))
    fmt.Printf("Debt: Offsetof: %d SizeOf: %d, Alignof: %d\n",
unsafe.Offsetof(cust.Debt), unsafe.Sizeof(cust.Debt),
unsafe.Alignof(cust.Debt))
```

```
    fmt.Printf("Discount: Offsetof: %d SizeOf: %d, Alignof: %d\n",
unsafe.Offsetof(cust.Discount), unsafe.Sizeof(cust.Discount),
unsafe.Alignof(cust.Discount))

    fmt.Printf("Align struct: %d\n", unsafe.Alignof(cust))
    fmt.Println("Struct SizeOf:", unsafe.Sizeof(cust))
}
```

Для каждого поля структуры мы вывели смещение, размер и выравнивание. Запустив код, мы увидим следующий вывод:

```
Bytes &[48]uint8{0x76, 0xfe, 0xc, 0x1, 0x0, 0x0, 0x0, 0x0, 0xc, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x19, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x98, 0x3a, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0xb8, 0xb, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
Name: Offsetof: 0 Sizeof 16 Alignof: 8
Age: Offsetof: 16 SizeOf: 2, Alignof: 2
Balance: Offsetof: 24 SizeOf: 8, Alignof: 8
Debt: Offsetof: 40 SizeOf: 8, Alignof: 8
Discount: Offsetof: 32 SizeOf: 1, Alignof: 1
Align struct: 8
Struct SizeOf: 48
```

Сначала идет байтовое представление нашей структуры, а далее - вся нужная нам информация по каждому полю. Если посмотрим на значения размера всех полей, то можем предположить, что размер нашей структуры - 35 байт. Но в самом конце вывода видим, что на самом деле структура занимает 48 байт памяти.
Теперь давайте посмотрим на значения выравнивания для всех полей и на байтовое представление структуры. Наглядно видно, как компилятор, выполняя выравнивание данных, заполняет память нулями. На примере поля Age видно, что его смещение 16 байт от начала структуры и размер 2 байта. Но смещение поля Balance уже 24 байта, вместо ожидаемых 18. То есть, фактически представление этого поля в памяти занимает 8 байт вместо 2.
Давайте посмотрим на поле Discount. Его размер составляет 1 байт. В байтовом представлении структуры это 32 элемент от начала структуры. Его значение 0x1. Передвинем его в нашей структуре и разместим после поля Age.

```
type Customer struct {
      Name string
      Age int16
      Discount bool
      Balance int
      Debt int
```

```
}
```

Запустим программу. Вывод немного изменится.

```
Bytes &[40]uint8{0x16, 0xfe, 0xc, 0x1, 0x0, 0x0, 0x0, 0x0, 0xc, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x19, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0,
0x98, 0x3a, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0xb8, 0xb, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0}
Name: Offsetof: 0 Sizeof 16 Alignof: 8
Age: Offsetof: 16 SizeOf: 2, Alignof: 2
Balance: Offsetof: 24 SizeOf: 8, Alignof: 8
Debt: Offsetof: 32 SizeOf: 8, Alignof: 8
Discount: Offsetof: 18 SizeOf: 1, Alignof: 1
Align struct: 8
Struct SizeOf: 40
```

Наша структура стала занимать уже 40 байт. И если взглянуть на смещение поля Discount, то увидим, что значение поля располагается следом за байтовым представлением поля Age без дополнительных пропусков между ними. То есть смещение 18 байт от начала структуры. И если посмотреть на байтовое представление структуры, то 18 элемент будет как раз значением Discount — 0x1.

На этом примере мы выполнили небольшую оптимизацию. Тут она не столь значительная и больше нужна как иллюстрация. Но на больших структурах данных и при разработке критичных к скорости алгоритмов такие оптимизации играют не последнюю роль.

unsafe.Pointer

Мы уже сталкивались с указателями и целочисленным типом uintptr, который достаточно большой, чтобы вместить в себя значение указателя. То есть представлением указателя является некоторое целое число.

Так зачем нам еще один указатель?

unsafe.Pointer — это указатель на произвольный тип данных. В то время как uintptr мы применяли в программах как указатель на тип (*int, *string и т.д.).

unsafe.Pointer содержит в себе точный адрес памяти, по которому лежат данные произвольной переменной.

Как и для обычного указателя, значением по умолчанию для типа unsafe.Pointer является nil.

То есть, как вы, наверное, уже понимаете, unsafe.Pointer позволяет обратиться к чистым данным в обход системы типов и при помощи преобразований представить эти данные в нужном вам типе. В этом кроется вся мощь и опасность пакета unsafe.

Адресная арифметика в Go заключается в преобразовании unsafe.Pointer к uintptr, выполнении математических операций с последующим преобразованием uintptr обратно к unsafe.Pointer. А получившийся unsafe.Pointer можно представить в любой тип.

Но не стоит вдохновляться раньше времени! Здесь вас поджидают некоторые неочевидные проблемы. Например, вы можете потерять контроль над связью между

указателем и данными, что может привести к непредсказуемым результатам. Пример ниже как раз иллюстрирует такую ситуацию.

```
package main

import (
    "fmt"
    "reflect"
    "unsafe"
)

func main() {
    ints := []int16{4,5,6,7,1,2,3,53}
    lenInts := len(ints)
    capInts := cap(ints)

    pt := uintptr(unsafe.Pointer(&ints[0]))

    ints = append(ints, 77)
    n := (*int16)(unsafe.Pointer(pt + unsafe.Sizeof(ints[0]) * 2))
    *n = 14

    fmt.Println(ints)        // ожидаем [4 5 14 7 1 2 3 53 77], а
получаем [4 5 6 7 1 2 3 53 77]

    sliceHeader := &reflect.SliceHeader{
        Data: pt,
        Len:  lenInts,
        Cap:  capInts,
    }
    oldInts := *(*[]int16)(unsafe.Pointer(sliceHeader))
    fmt.Println(oldInts)      //[4 5 14 7 1 2 3 53]
}
```

Давайте разберемся в том, что же происходит в коде. Есть слайс []int16, в котором есть некоторое количество элементов. Нашей целью будет изменить значение третьего элемента слайса, используя пакет unsafe.
Приступим к реализации. Сохраним исходные параметры слайса в переменные lenInts и capInts. Затем получим указатель на нулевой элемент слайса, то есть на начало самих данных, а не на структуру. И, преобразуя указатель в uintptr, сохраним в переменную pt. uintptr нам понадобится для выполнения арифметики смещения. Здесь мы совершаем главную ошибку, но об этом мы узнаем немного позже.

Следующим шагом мы смоделируем изменение наших исходных данных и добавим в исходный слайс один элемент.
Далее, чтобы изменить какой-то элемент, нам нужно получить на него указатель. Теперь разберемся, что же делают эти строки:

```
n := (*int16)(unsafe.Pointer(pt + unsafe.Sizeof(ints[0]) * 2))
*n = 14
```

К указателю на первый элемент слайса, который мы сохранили в переменной pt, добавляем смещение на 4 байта. То есть, получаем размер одного элемента слайса ints и умножаем на 2. Далее получаем новый указатель на адрес и преобразовываем к указателю на тип int16. Теперь в переменной n сохранен указатель на нужный нам элемент слайса и через этот указатель устанавливаем новое значение.
Но запуск программы показывает нам совсем не то, что мы ожидаем увидеть. Вместо слайса [4 5 14 7 1 2 3 53 77] , который мы увеличили и изменили третий элемент, мы получаем [4 5 6 7 1 2 3 53 77], то есть увеличенный, но с исходным значением третьего элемента.
Это случилось потому, что при увеличении размера слайса изменилась его вместимость, компилятор выделил новую память нужного размера и переместил туда наш слайс. Но при этом старые значения остались, просто на эту область памяти пока никто не ссылается.
Далее проверим наше утверждение. Соберем новый слайс. Он будет указывать на область памяти, которую занимал наш исходный слайс.

```
sliceHeader := &reflect.SliceHeader{
    Data: pt,
    Len:  lenInts,
    Cap:  capInts,
}

oldInts := *(*[]int16)(unsafe.Pointer(sliceHeader))
```

При помощи пакета reflect создаем новую структуру слайса и указываем атрибуты исходного слайса: указатель на исходный массив, длину и вместимость. Затем получаем указатель на новый слайс и преобразовываем его к указателю типа *[]int16. И теперь мы видим, в какой области памяти на самом деле мы изменили данные.
Наша ошибка, которую мы допустили в самом начале, заключается в том, что мы один раз вычислили указатель на данные и сохранили его в переменную. Но при перемещении данных не произошло обновления указателя. Кроме этого, преобразование указателя в uinptr разрывает связь между указателем и значением, поэтому слайс ints может быть уничтожен сборщиком мусора. И попытка обращения по сохраненному указателю uintptr также может привести к непредсказуемым последствиям.
Правильным решением было бы делать преобразования к uintptr при необходимости.

```
package main
```

```
import (
    "fmt"
    "unsafe"
    "reflect"
)

func main() {
    ints := []int16{4,5,6,7,1,2,3,53}
    lenInts := len(ints)
    capInts := cap(ints)

    pt := unsafe.Pointer(&ints[0])

    ints = append(ints, 77)
    n := (*int16)(unsafe.Pointer(uintptr(unsafe.Pointer(&ints[0])) +
unsafe.Sizeof(ints[0]) * 2))
    *n = 14

    fmt.Println(ints)        // [4 5 14 7 1 2 3 53 77]

    sliceHeader := &reflect.SliceHeader{
        Data: uintptr(pt),
        Len:  lenInts,
        Cap:  capInts,
    }
    oldInts := *(*[]int16)(unsafe.Pointer(sliceHeader))
    fmt.Println(oldInts)      //[4 5 6 7 1 2 3 53]
}
```

Заключение
Пакет unsafe - это инструмент, который позволяет обойти безопасность типов и на самом низком уровне реализовывать алгоритмы. Но стоить помнить, что unsafe не предназначен для реализации бизнес-логики, его не нужно применять для перебора сделок в аккаунте пользователя или для обработки данных с формы ввода. Это возможность максимально тонкой оптимизации узких мест системы.

Полезные ссылки:

- Package unsafe
- Type-Unsafe Pointers

- Безопасное использование unsafe

Задание:

1. Сделайте форк проекта module10_04 в группу golang_users_repos/<your_gitlab_id> и создайте ветку 04_task.
2. В файле main.go определена структура Lead с некоторыми данными. В файл main.go допишите код, в котором реализуйте следующее:
- Измените значение поля Budget, используя unsafe.Pointer и смещение полей.
- При помощи указателя на элементы слайса Items и пакета reflect создайте новый слайс. Добавьте в него произвольное количество элементов и замените исходный Items в структуре lead.
3. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 04_task.