Слабые Указатели в Go

января 2, 2025 · 5 минут · German Gorelkin

Перевод/заметки Weak Pointers in Go: Why They Matter Now

Слабый указатель(weak pointer) - это, по сути, способ сослаться на участок памяти без его блокировки. Благодаря этому сборщик мусора может удалить его, если больше никто не удерживает этот блок активно.

В языке Go существует концепция слабых указателей, которая является частью пакета `weak`, тесно связанного с runtime Go. Любопытно, что ранее этот пакет был в основном предназначен для внутреннего использования, однако в последнее время его планируют сделать общедоступным, о чём свидетельствует

это proposal.

Главное преимущество слабых указателей заключается в их безопасности. Когда память, на которую они ссылаются, освобождается, слабый указатель автоматически становится `nil`, что исключает риск случайного доступа к освобожденной памяти. Если же вам необходимо сохранить ссылку на память, вы можете преобразовать слабый указатель в **сильный (strong)**.

Работать с ними сложнее, чем с обычными указателями. В любой момент слабый указатель может стать `nil`, если память, на которую он ссылается, будет освобождена. Это может произойти, если ни один сильный указатель не сохраняет ссылку на эту память. Поэтому крайне важно всегда проверять, не стал ли только что созданный сильный указатель `nil` после преобразования из слабого.

Очистка происходит не сразу. Даже если нет ссылок на память, то время очистки зависит от работы сборщика мусора.

Пакет `weak` предоставляет две основные возможности:

- `weak.Make`: создает слабый указатель из сильного.
- `weak.Pointer[T].Strong`: преобразует слабый указатель обратно в сильный.

```
type T struct {
  a int
  b int
}

func main() {
  a := new(string)
  println("original:", a)

  // make a weak pointer
  weakA := weak.Make(a)

  runtime.GC()

  // use weakA
  strongA := weakA.Strong()
  println("strong:", strongA, a)
```

```
  runtime.GC()

  // use weakA again
  strongA = weakA.Strong()
  println("strong:", strongA)
}

// Output:
// original: 0x1400010c670
// strong: 0x1400010c670 0x1400010c670
// strong: 0x0
```

- После первой сборки мусора, вызванной командой `runtime.GC()`, слабый указатель `weakA` все ещё указывает на область памяти, поскольку мы продолжаем использовать переменную `a` в строке `println(«strong:», strongA, a)`. Память не может быть освобождена, так как она используется.
- Однако, когда будет запущена вторая сборка мусора, сильный указатель (`a`) уже не будет использоваться. Это позволяет сборщику мусора безопасно освободить память, что приведёт к тому, что функция `weakA.Strong()` будет возвращать `nil`.

Если вы попытаетесь запустить этот код не с `*string`, а с `*int`, `*bool` или другим типом, то можете столкнуться с неожиданным поведением: последний `strong` вывод может отличаться от `nil`.

Это связано с тем, как Go обрабатывает «*tiny objects*», такие как `int`, `bool`, `float32`, `float64` и т.д. Эти типы алоцируются как маленькие объекты, и даже если они технически не используются, сборщик мусора не всегда очищает их сразу.

Слабые указатели могут быть очень полезны для управления памятью в определенных сценариях.

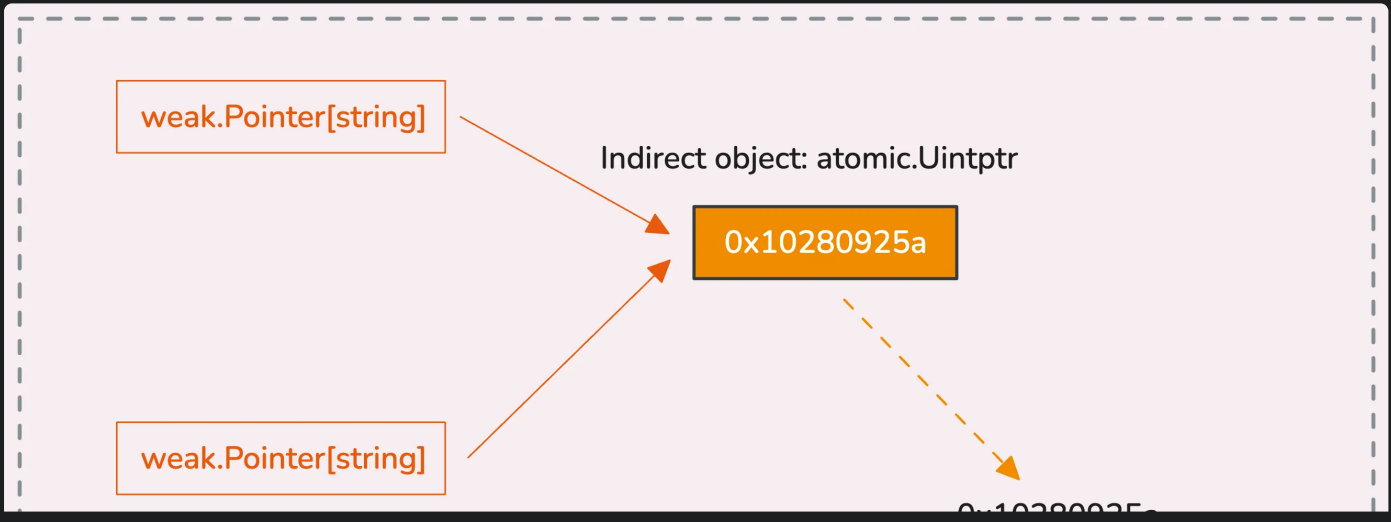
- Например, они отлично подходят для *canonicalization maps* - ситуаций, когда вы хотите хранить только одну копию части данных.
- Другой случай - когда вы хотите, чтобы срок жизни некоторой памяти соответствовал сроку жизни другого объекта, подобно тому, как работает WeakMap в JavaScript. WeakMap позволяют автоматически очищать объекты, когда они больше не используются.

Итак, главное преимущество слабых указателей заключается в том, что они позволяют вам сообщить сборщику мусора: «*Эй, ты можешь избавиться от этого ресурса, если его никто не использует, я всегда смогу воссоздать его позже*». Это особенно полезно для объектов, которые занимают много памяти, но не должны оставаться в ней, если их не используют активно.

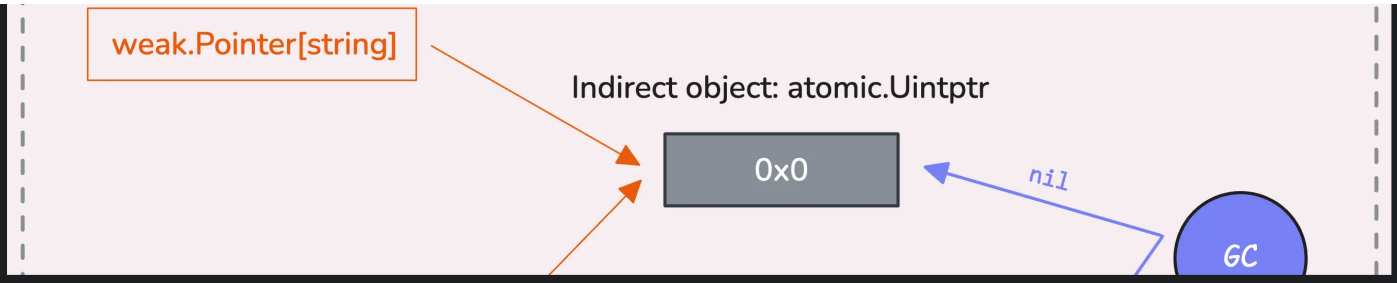
Как работают слабые указатели?

Интересно, что слабые указатели на самом деле не указывают непосредственно на память, на которую ссылаются. Вместо этого они представляют собой простые структуры, которые содержат «indirection object». Этот объект крошечный, всего 8 байт, и он указывает на реальный объект памяти.

```
type Pointer[T any] struct {
    u unsafe.Pointer
}
```



Эта настройка позволяет сборщику мусора эффективно удалять слабые указатели на конкретный объект за один раз. Когда сборщик решает освободить память, ему нужно лишь установить указатель в объекте перенаправления на `nil` (или `0x0`). Ему не требуется обновлять каждый слабый указатель по отдельности.



Кроме того, эта конструкция позволяет выполнять проверку на равенство с помощью оператора ==. Слабые указатели, созданные на основе одного и того же исходного указателя, будут считаться «равными» даже после того, как объект, на который они ссылаются, будет уничтожен.

```
func main() {
    a := new(string)

    // make a weak pointers
    weakA := weak.Make(a)
    weakA2 := weak.Make(a)

    println("Before GC - Equality check:", weakA == weakA2)

    runtime.GC()

    // Test their equality
    println("After GC - Strong:", weakA.Strong(), weakA2.Strong())
    println("After GC - Equality check:", weakA == weakA2)
}

// Before GC - Equality check: true
// After GC - Strong: 0x0 0x0
// After GC - Equality check: true
```

Это работает потому, что все слабые указатели, которые ссылаются на один и тот же исходный объект, используют один и тот же объект перенаправления. Когда вы вызываете `weak.Make`, если у объекта уже есть связанный с ним слабый указатель, существующий объект перенаправления повторно используется вместо

создания нового.

> *Подождите, а не слишком ли расточительно использовать 8 байт для объекта перенаправления?*

Слабые указатели обычно применяются в ситуациях, когда основной задачей является экономия памяти. Например, в canonicalization maps, где мы удаляем дубликаты, сохраняя только одну копию каждого уникального фрагмента данных, мы уже значительно уменьшаем объем используемой памяти за счёт устранения избыточности.

Тем не менее, если вы применяете слабые указатели в ситуации, когда у вас много уникальных элементов и мало повторяющихся, вы можете столкнуться с тем, что используете больше памяти, чем ожидалось. Поэтому при решении вопроса о том, подходят ли слабые указатели для вашей задачи, необходимо учитывать специфику конкретного случая использования.

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Как устроен map в go

СЛЕДУЮЩАЯ »
Go sync.Cond - самый недооцененный механизм синхронизации