



uvelichitel 29 марта 2016 в 12:54

О функциональности Go

Функциональное программирование, Программирование, Go

Насколько объектно Go ориентирован многократно и эмоционально обсуждалось. Попробуем теперь оценить насколько он функционален. Заметим сразу, оптимизацию хвостовой рекурсии компилятор не делает. Почему бы? «Это не нужно в языке с циклами. Когда программист рекурсивный код, он хочет представлять стек вызовов или он пишет цикл.» — замечает в переписке Russ Cox. В языке зато есть полноценные `lambda`, `closure`, рекурсивные типы и ряд особенностей. Попробуем их применить функциональным манером. Примеры покажутся синтетичными, что во первых написаны немедленно исполняемыми в песочнице и написаны на процедурном все же языке во вторых. Предполагаю знакомство как с Go так и с функциональным программированием, разъяснений мало но код комментируется.

Closure, замыкание реализовано в языке в классической и полной мере.

Например ленивую рекурсивную последовательность можно получить так

```
func produce(source int, permutation func(int) int) func() int {
    return func() int {
        source = permutation(source) //замыкание source
        return source
    }
}
```

Простой вариатор для псевдослучайных чисел

```
func mutate(j int) int {
    return (1664525*j + 1013904223) % 2147483647
}
```

И вот наш генератор случайных чисел

```
next := produce(1, mutate)
next()
```

Работающий пример

Попробовать в песочнице

Currying. каррирование, применение функции к одному из аргументов в общем случае не реализовано. Частные задачи однако решаются. Например функция отложенного вызова стандартной библиотеки `time` имеет сигнатуру `func AfterFunc(d Duration, f func()) *Timer` принимает аргументом `func()`, а мы бы хотели передать нечто более параметризованное `func(arg MyType)`. И мы можем это сделать так

```
type MyType string //объявление типа
func (arg MyType) JustPrint(){ //объявление метода
    fmt.Println(arg)
}
```

метод в Go это функция принимающая первым аргументом своего бенефициара

Выражение `MyType.JustPrint` даст нам эту функцию с сигнатурой `func(arg MyType)`, которую мы можем применить к аргументу `MyType.JustPrint(«Съешь меня»)`

Напротив выражение `arg.JustPrint` даст нам функцию `JustPrint` примененную к `arg` с сигнатурой `func()` которую мы и можем передать нашему будильнику

```
timer := time.AfterFunc(50 * time.Millisecond, arg.JustPrint)
```

Работающий пример

Попробовать в песочнице.

Continuation, продолжение как первоклассный объект не реализован с той элегантностью что в `scheme`. Есть между тем встроенная функция `panic()`, приблизительный аналог `long_jump` способная прервать вычисления и вернуть при этом достигнутый результат(например ошибку) место откуда был сделан вызов. Конструкцию `panic()`, `defer recover()` кроме обработки исключений можно применить например для сквозного выхода из зашедшей слишком глубоко рекурсии(что заметим и делается в пакете `encoding.json`). В этом смысле конструкция первоклассна исключительно. Выход из ненужной рекурсии, стоит подчеркнуть, это классическое применение continuation.

Вот прямолинейная, не оптимизированная(не применять в production!!) рекурсивная функция отдающая n-ное число Фибоначчи как сумму предыдущих

```
func Fib(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    first := Fib(n - 1)
    second := Fib(n - 2)
    if first > max {           //Если числа стали излишне велики
        panic(second)         //то здесь вычисления прерываются со сбором урожая
    }
    return first + second
}
```

Так мы ее вызовем с продолжением(call/cc) желая получить n-ное число Фибоначчи, если только оно не больше max

```
var max int = 200
func CallFib(n int) (res int) {
    defer func() {
        if r := recover(); r != nil { //восстановление продолжения
            res = r.(int)              //плоды трудов
        }
    }()
    res = Fib(n)
    return
}
```

Работающий пример.

Попробовать в песочнице.

Монады в понимании Haskell процедурному языку просто не нужны. В Go между тем вполне разрешены рекурсивные объявления типов, а как раз и полагают монады видом структурной рекурсии. Rob Pike предложил следующее определение state machine, конечного автомата

```
type stateFn func(Machine) stateFn
```

где состояние это функция машины производящая действия и возвращающая новое состояние.

Работа такой машины проста

```
func run(m Machine) {
    for state := start; state != nil; {
        state = state(m)
    }
}
```

Разве не напоминает Haskell State Monad.

Напишем минимальный парсер, а для чего же еще нужны state machine, выбирающий числа из входящего потока.

```
type stateFn func(*lexer) stateFn
type lexer struct {
    *bufio.Reader //машине нужна лента
}
```

Нам достаточно всего двух состояний

```
func lexText(l *lexer) stateFn {
    for r, _, err := l.ReadRune(); err != io.EOF; r, _, err = l.ReadRune() {
        if '0' <= r && r <= '9' { //если попала цифра
            l.UnreadRune()
            return lexNumber //переход состояния
        }
    }
    return nil // Стоп машина.
}

func lexNumber(l *lexer) stateFn {
    var s string
```

```

    for r, _, err := l.ReadRune(); err != io.EOF; r, _, err = l.ReadRune() {
        if '0' > r || r > '9' { //если не цифра
            num, _ := strconv.Atoi(s)
            return lexText //переход состояния
        }
        s += string(r)
    }
    num, _ := strconv.Atoi(s)
    return nil // Стоп машина.
}

```

Работающий пример.

Попробовать в песочнице.

Реактивное программирование сложно формально описать. Это что то о потоках и сигналах. В Go есть то и другое. Стандартная библиотека предлагает интерфейсы *io.Reader* и *io.Writer* имеющие методы *Read()* и *Write()* соответственно и достаточно стройно отражающие идею по Файл и сетевое соединение к примеру реализуют оба интерфейса. Использовать интерфейсы можно безотносительно к источнику данных, скажем

```

Decoder = NewDecoder(r io.Reader)
err = Decoder.Decode(Message)

```

будет единообразно кодировать файл или например сетевое соединение.

Идея сигналов воплощена в синтаксисе языка. Тип *chan* (channel) оснащен оператором *<—* передачи сообщений, а уникальная конструкция *select{ case <— chan }* позволяет выбрать готовый к передаче канал из нескольких.

Напишем совсем простой миксер потоков.

В качестве входных потоком возьмем просто строки. (Мы условились делать примеры немедленно исполняемыми в песочнице, что ограничивает выбор. Читать из сетевого соединения было бы интересней. И код может практически без изменений.)

```

reader1 := strings.NewReader("ла ла ла ла ла ла ла")
reader2 := strings.NewReader("фа фа фа фа фа фа фа")

```

Выходным приемом стандартный поток вывода

```

writer := os.Stdout

```

В качестве управляющих сигналов используем канал таймера.

```

stop := time.After(10000 * time.Millisecond)
tick := time.Tick(150 * time.Millisecond)
tack := time.Tick(200 * time.Millisecond)

```

И весь наш миксер

```

select {
case <-tick:
    io.CopyN(writer, reader1, 5)
case <-tack:
    io.CopyN(writer, reader2, 5)
case <-stop:
    return
}

```

Работающий пример.

Попробовать в песочнице.

Метки: go, функциональное программирование

Добавить метки

↑ +16 ↓ 73 11,8k 6



↑ 25,0 ↓
Карма

0,0
Рейтинг

6
Подписчики

✉ Написать

✍ Подписать

Костарев Илья @uvelichitel

Пользователь

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

22 марта 2017 в 18:59

Функциональное программирование и с++ на практике

↑ +7

👁 10,8k

🔖 64

💬 15

19 марта 2017 в 16:50

О функциональном программировании в фронтенде

↑ +11

👁 23,4k

🔖 138

💬 197

16 сентября 2016 в 12:49






Жаргон функционального программирования

↑ +83

👁 56,2k


🔖 527

💬 113

ВАКАНСИИ		Мой к
	Senior Frontend разработчик (React) CORE . Construct Online Resources for Education · Москва · Возможна удаленная работа	от 100 (
	React Native / React.JS программист удалённо Start Mobile · Возможна удаленная работа	от 55 (
	Backend-разработчик, Node.js Pantini Inc. · Москва	до 200 (
	Project manager FunCorp · Москва	до 200 (
	Программист 1С Аксон · Москва	от 130 000 до 150 (
Все вакансии		

Комментарии 6

Отслеживать новые в ☐ почте ☐




evocatus 29.03.16 в 16:06

🔗 📌

↑

Отсутствие оптимизации хвостовой рекурсии в Go расстроило. Лично для меня это один из главных минусов Python.
Почему рекурсия зло можно почитать здесь: <https://habrahabr.ru/post/256351/>

Ответить




evocatus 29.03.16 в 16:10

🔗 📌 ↻

↑

Притом в Lua (откуда Go взял немало особенностей синтаксиса, судя по всему) хвостовая рекурсия оптимизируется.

Ответить



Neftedollar 29.03.16 в 18:16

🔗 📌 ↻


↑

Очень странная статья по ссылке.
Половину примеров можно привести к хвостовой рекурсии, которую компилятор либо оптимизирует, либо вы ею не будете пользоваться, особенн

<https://habr.com/post/280210/>

вы знаете что такое хвостовая рекурсия.
Вторую же половину читабельнее, красивее и проще чем не хвостовой рекурсией описать сложно.

Ответить

 **beduin01** 29.03.16 в 16:51  

↑

Даже богом забытый Nim на фоне Go функциональнее смотрится.

Ответить

 **uvelichitel** 29.03.16 в 17:30    

↑

На плашке Go, где сейчас написано "simple, reliable and efficient" раньше было "pragmatic, concise and orthogonal", а еще раньше в самом начале "procedural language for system programming". В общем то "functional, multiparadigm" там не было никогда.

Ответить

 **Googolplex** 29.03.16 в 21:41  




↑

Для нормального функционального программирования необходимы (но не достаточны!) либо полноценный параметрический полиморфизм (т.е. дженерики; например, это языки ML-семейства), либо полная динамичность (лиспы, эрланг). Go не имеет ни того, ни другого, и поэтому максимум, что можно сказать, это то, что в нём есть функциональные элементы.

Но это, безусловно, не значит, что язык плохой — просто он задумывался для другого, и его основные практики и подходы тоже другие.

Ответить

Написать комментарий

B / U         *

Предпросмотр

Отправить

☐ Markdown

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц


Отсутствие дискриминации – это основная ценность open source

↑ +67  16,4k  28  120

Почему мне не перезвонили?

↑ +24  23,3k  68  68

DEFCON 17. Взлом 400 000 паролей, или как объяснить соседу по комнате, почему счёт за электричество увеличился. Часть 1

↑ +16  6,3k  43  3

Как Microsoft спрятала целый сервер и как его найти

↑ +3  6,1k  14  5

В чём важность $196\,884 = 196\,883 + 1$? Как это объяснить на пальцах?

↑ +52  25,7k  119  101

Профиль	Публикации	Правила	Реклама
Трекер	Хабы	Помощь	Тарифы
Диалоги	Компании	Документация	Контент
Настройки	Пользователи	Соглашение	Семинары
ППА	Песочница	Конфиденциальность	

