Конкурентность на базе горутин

Когда `responseSize` вызывает `http.Get`, вашей программе приходится ожидать ответа сайта. Во время ожидания она не делает ничего полезного.

Возможно, другой программе придется дожидаться пользовательского ввода. А еще одной программе придется ждать чтения данных из файла. Существует множество ситуаций, в которых программам приходится просто простаивать в ожидании.

Горутины обеспечивают возможность конкурентности: приостановки одной задачи для работы над другими задачами. А в других ситуациях они позволяют реализовать параллелизм: одновременную работу над несколькими задачами!

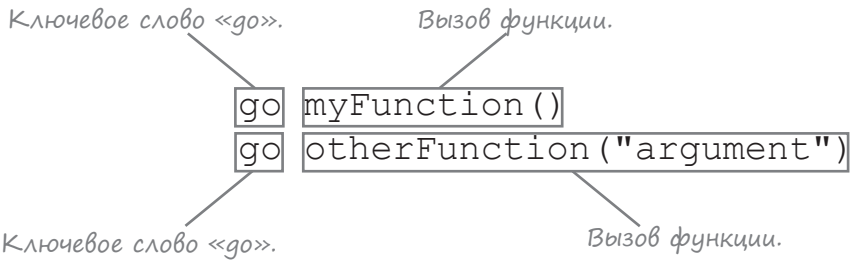
Конкурентность позволяет программе приостановить одну задачу и работать над другими задачами. Программа, ожидающая ввода от пользователя, может выполнять другие действия в фоновом режиме. Во время чтения из файла программа может обновлять индикатор прогресса. Наша программа `responseSize` может выдавать другие сетевые запросы, ожидая завершения первого запроса.

Если программа написана с поддержкой конкурентности, она также может поддерживать **параллельное выполнение**: *одновременное* выполнение задач. Компьютер с одним процессором может выполнять только одну задачу за раз. Тем не менее большинство современных компьютеров содержит несколько процессоров (или один процессор с несколькими ядрами). Ваш компьютер может распределить несколько конкурентных задач между несколькими процессорами, чтобы выполнять их одновременно. (Вам редко приходится управлять ими вручную: операционная система обычно делает все за вас.)

Разбиение больших задач на меньшие подзадачи, которые могут выполняться конкурентно, иногда приводит к существенному приросту скорости ваших программ.

В Go конкурентно выполняемые задачи называются **горутинами**. В других языках программирования существует аналогичная концепция *потоков*, но горутины расходуют меньше компьютерной памяти, чем потоки, а также быстрее запускаются и останавливаются, а это означает, что вы можете запускать больше горутин одновременно.

Кроме того, горутины проще в использовании. Для запуска новой горутины используется `go`-команда — обычный вызов функции или метода, перед которым находится ключевое слово `go`:



Обратите внимание: мы говорим «*другую* горутину». Функция `main` каждой программы Go запускается с помощью горутины, так что в каждой программе Go выполняется по крайней мере одна горутина. Выходит, мы все это время пользовались горутинами, не подозревая об этом!

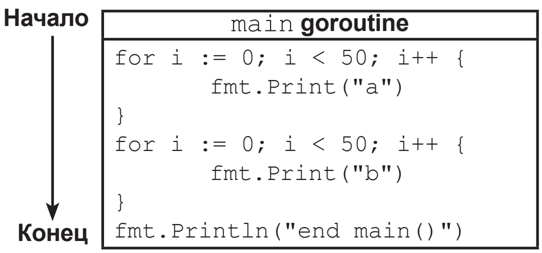
Использование горутин

Следующая программа вызывает функции по одной. В ней используется цикл, который выводит строку "a" 50 раз, а функция b выводит строку "b" 50 раз. Функция main вызывает a, затем b, а потом выводит сообщение при завершении.

```
package main

import "fmt"

func a() {
	for i := 0; i < 50; i++ {
		fmt.Print("a")
	}
}

func b() {
	for i := 0; i < 50; i++ {
		fmt.Print("b")
	}
}

func main() {
	a()
	b()
	fmt.Println("end main()")
}
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbend main()
```

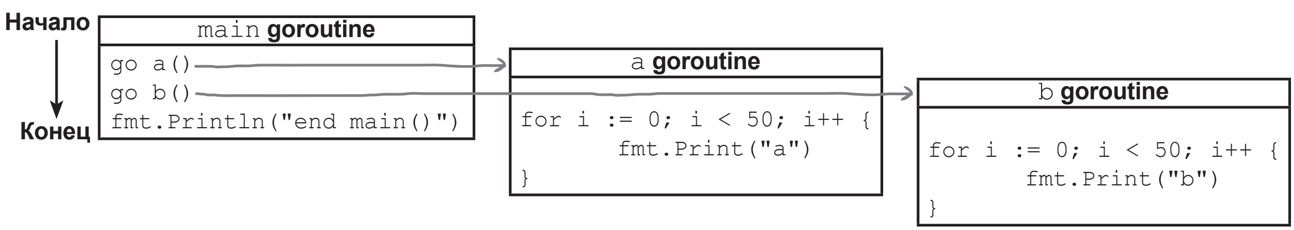
Все происходит так, как если бы функция main содержала весь код функции a, за которым следовал бы весь код функции b и собственный код main:



Чтобы запустить функции a и b в новых горутинах, достаточно добавить ключевое слово go перед вызовами функций:

```
func main() {
	go a()
	go b()
	fmt.Println("end main()")
}
```

Тогда новые горутины будут конкурентно выполняться в функции main:



Использование горутин (продолжение)

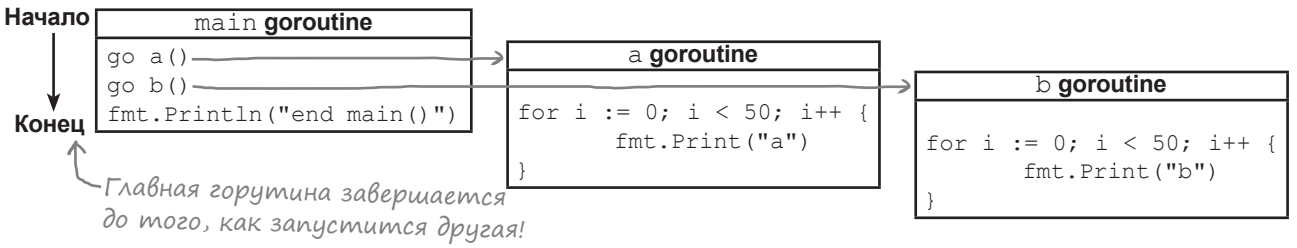
Но если запустить программу сейчас, то единственный результат, который мы увидим, будет выведен функцией `Println` в конце `main`. А вот функции `a` и `b` ничего не выводят!

```
func main() {
        go a()
        go b()
        fmt.Println("end main()")
}
```

Нет вывода функций «a» и «b»?

```
end main()
```

Проблема вот в чем: программы Go перестают выполняться сразу же после завершения горутины `main` (той, которая вызывает функцию `main`), даже если другие горутины продолжают выполняться. Наша функция `main` завершается до того, как код функций `a` и `b` получит возможность выполниться.



Горутина `main` должна продолжать выполнение до того, как горутины функций `a` и `b` смогут завершиться. Чтобы правильно реализовать эту последовательность выполнения, понадобится еще одно средство Go — так называемые *каналы*, но этот механизм будет рассматриваться чуть позже в этой главе. Итак, пока мы просто приостановим горутину `main` на заданный период времени, чтобы смогли выполниться другие горутины.

Для этого мы воспользуемся функцией `Sleep` из пакета `time`. Эта функция приостанавливает текущую горутину на заданный промежуток времени. Вызов `time.Sleep(time.Second)` из функции `main` заставит горутину `main` приостановить выполнение на 1 секунду.

```
func main() {
        go a()
        go b()
        time.Sleep(time.Second)
        fmt.Println("end main()")
}
```

Горутина main приостанавливается на 1 секунду.

Дает другим горутинам достаточно времени для выполнения.

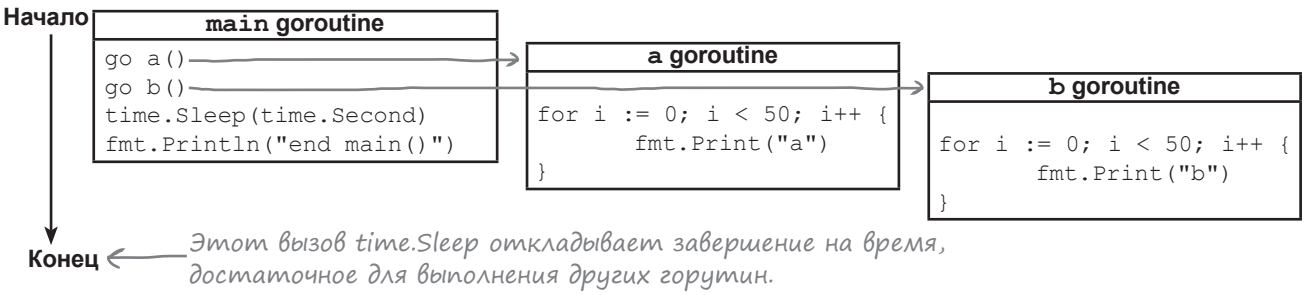
```
aaaaaaaaaaaaaaaaaaaaaaabbbbbaaa
aaaaaaaabbbbbbbbbbbaaaaaaaaaaa
abbaaaaabbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbend main()
```

Когда time.Sleep вернет управление, горутина main завершит выполнение.

Если повторно запустить программу, вы снова увидите выводы функций `a` и `b`, так как их горутины получат возможность выполниться. Вывод этих функций будет чередоваться, так как программа переключается между двумя горутинами. (Закономерность чередования может отличаться от той, которая показана здесь.) Когда Go-функция `main` снова активизируется, она вызовет `fmt.Println` и завершится.

Использование горутин (продолжение)

Вызов `time.Sleep` в главной горутине дает более чем достаточно времени для выполнения функций `a` и `b`.



Использование горутин с функцией responseSize

Программа для вывода размеров веб-страниц легко преобразуется для использования горутин. Для этого понадобится совсем немного — добавить ключевое слово `go` перед каждым вызовом `responseSize`.

Чтобы горутина `main` не завершилась перед тем, как горутины `responseSize` смогут завершиться, также нужно будет добавить вызов `time.Sleep` в функцию `main`.

```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "time"                    ← Добавляется пакет «time».
)

func main() {
        go responseSize("https://example.com/")
        go responseSize("https://golang.org/")
        go responseSize("https://golang.org/doc")
        time.Sleep(5 * time.Second)
}

func responseSize(url string) {
        fmt.Println("Getting", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(len(body))
}
```

Вызовы responseSize преобразуются в go-команды.

Пятисекундная пауза.

Впрочем, приостановка всего на 1 секунду может оказаться недостаточной для завершения сетевых запросов. Вызов `time.Sleep(5 * time.Second)` обеспечит приостановку горутины на 5 секунд. (Если вы попробуете запустить эту программу с медленной или перегруженной сетью, возможно, это время придется увеличить.)