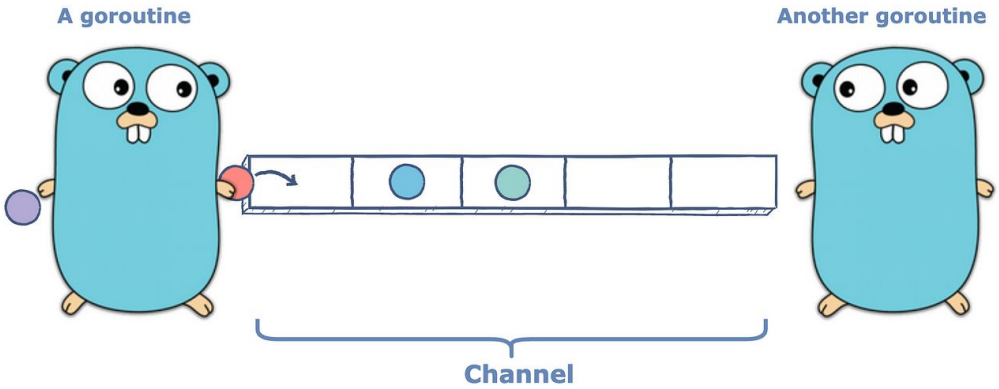
# 2. Golang concurrency

# Goroutines

- В ОС есть потоки
  - kernel space, syscall
  - создать/удалить поток затратно
- легкий поток, управляемый на уровне Go runtime
  - то есть, на уровне user-space
  - меньше места
  - быстрее переключаться
  - можно создать десятки тысяч горутин
- `go f(x, y, z)`
- https://go.dev/tour/concurrency/1
  - код в горутине исполняется одновременно с кодом снаружи горутины

# Каналы

- средства синхронизации между горутинами
- бывают буферизованные и не буферизованные
  - `make(chan int, 2)`
  - `make(chan int)`
- пример с буферизованным каналом https://go.dev/tour/concurrency/3
- пример с небуферизованным
  - https://go.dev/play/p/o9gF8kn4Eyz
  - https://go.dev/play/p/NYBMSITLuHY



# Операции с каналами

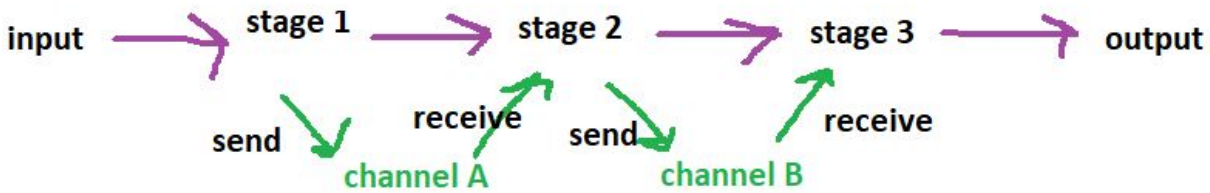
- В канал можно писать, из канала можно читать
- Канал можно закрыть с помощью `close`
- Канал можно читать через range
  - значения из канала будут приходить до тех пор, пока канал не закроется
  - после чего программа выйдет из `range`
  - если никто канал не закрыл, будем вечно ждать значений из range
- https://go.dev/tour/concurrency/4

# Особенности поведения каналов

- Нельзя читать/писать из nil канала https://go.dev/play/p/xwtnm0sdpXq
- Нельзя писать в закрытый канал https://go.dev/play/p/yWf9IeAqtbt
  - хорошей практикой считается, чтобы закрывал канал тот, кто в него пишет
  - и никогда не закрывал читатель канала
    - потому что писатель не узнает про то, что канал закрыт
- Буферизованный канал с большим буфером – неэффективно
  - make(chan int, 1000) – все это хранить в памяти?
- Если канал закрыт, чтение из него неблокирующее
  - даже если канал небуферизованный
  - https://go.dev/play/p/gEz8AdeDtdE
  - ok = false

# Паттерны: pipeline (конвейер)

- Можно обрабатывать данные последовательно:
  - читать данные из канала X
  - применять к данным операцию
  - результат класть в канал Y
  - применять к данным операцию
  - результат класть в канал Z
  - и т.д. …
- https://go.dev/blog/pipelines
- https://go.dev/play/p/MW84WD2gEhG

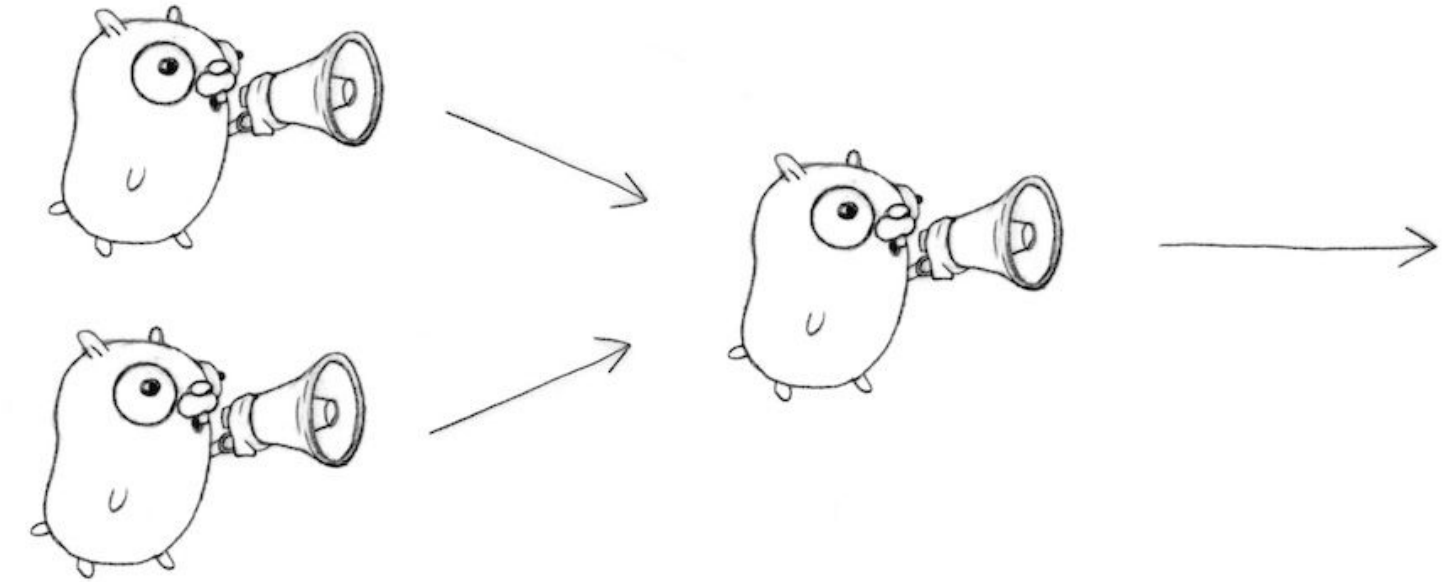


# Паттерны: генератор

- https://go.dev/talks/2012/concurrency.slide#25
- Когда хотим создать источник данных:
  - создаем данные
  - выдаем наружу канал
  - читатель канала получает данные
- https://go.dev/play/p/Ugt-Bx6sByb

# Паттерны: Fan-in

- https://go.dev/blog/pipelines ; https://go.dev/talks/2012/concurrency.slide#27
- Когда данные из нескольких каналов хотим слить в один
- https://go.dev/play/p/op9CV3gxj5k



# Select

- Когда нужно подождать результата нескольких операций
  - например, результата вычисления выражения или таймаута/сигнала завершения программы
  - заблокируется на ожидании до тех пор, пока не получит результат хотя бы одной из операций
- select https://go.dev/tour/concurrency/5
- select с default https://go.dev/tour/concurrency/6
  - когда хотим не ждать и получить результат операции, если он готов
  - а если не хотим – идти дальше

# Паттерны: explicit cancellation

- https://go.dev/blog/pipelines
- Нужно остановить программу, в которой запущено много горутин
  - как остановить горутины?
  - оставить их выполняться, пока ОС не убьет весь процесс и горутины вместе с ним – плохо
    - так как гипотетически ОС может убить не все
    - можно не разблокировать какие-то ресурсы
    - => повисшие процессы
- https://go.dev/play/p/6pONu6hK54x

# Стандартная библиотека: sync

- http://pkg.go.dev/sync#pkg-overview
- waitGroup
  - когда есть N каких-то асинхронных операций (тасок), и нужно дождаться, пока все они закончатся
- once
  - когда что-то нужно сделать строго один раз
- mutex
  - для
  - реализует интерфейс Locker
- еще есть:
  - RWMutex, map
  - pool, cond

# WaitGroup

- https://pkg.go.dev/sync#WaitGroup
- три метода:
  - `Add(delta int)` – установить счетчик горутин, которые надо ждать, равным delta
  - `Done()` – уменьшить счетчик на один
  - `Wait()` – ждать, пока счетчик не станет 0
- https://gobyexample.com/waitgroups
- счетчик не может становиться < 0 :
  - https://go.dev/play/p/Zuy6S7I8K3A
  - https://go.dev/play/p/HFEiKukp_zJ

# Once

- https://pkg.go.dev/sync#Once
- один метод:
  - `Do(f func())`
  - если много раз вызовем Do(), f func() выполнится только при первом вызове
- https://go.dev/play/p/rmPXf540Qof
- может быть полезно при инициализации чего-то:
  - если нужно один раз проинициализировать соединение с базой данных
  - один раз прочитать конфигурационный файл приложения при старте

# Мьютексы (замки)

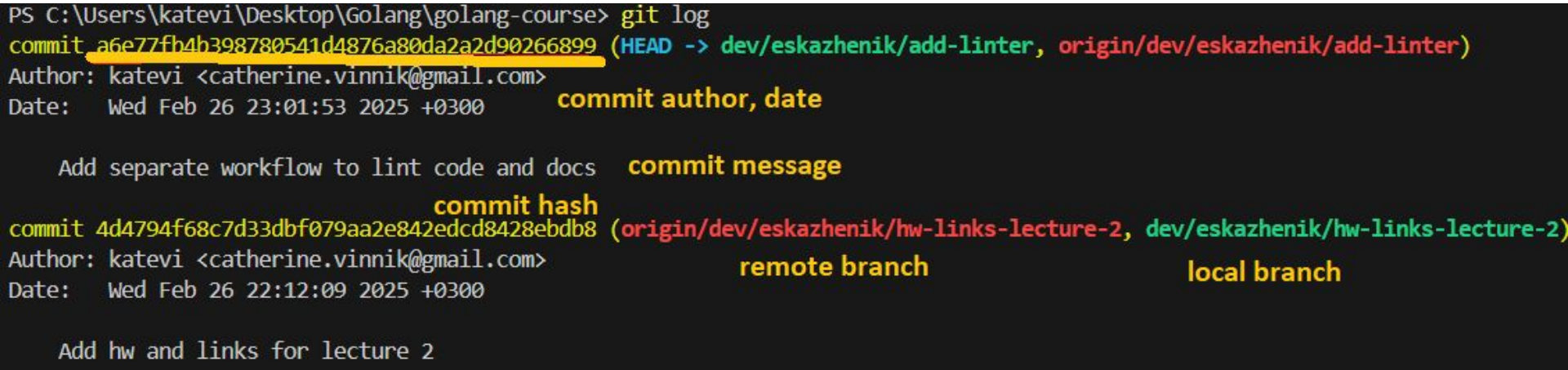
- https://pkg.go.dev/sync#Mutex
- можем из разных горутин пытаться изменить одни те же данные
  - => конкурентная модификация, гонка
  - изменения могут потеряться https://go.dev/play/p/9DydWidl9JV
  - данные могут стать неконсистентными https://go.dev/play/p/7LWALpRymik
- https://gobyexample.com/mutexes
- три метода
  - Lock() – взять блокировку
  - Unlock() – снять блокировку
  - TryLock() – обычно не нужен :)
- Unlock() всегда должен быть после Lock()
  https://go.dev/play/p/tadZxzxbYkh

# Git & GitHub

- Много разработчиков пишут код – как его объединять?
  - что-то написали, но поняли, что надо вернуть прошлую версию?
- Git – система контроля версий https://git-scm.com/book/en/v2
- Основная ветка (master)
  - разработчики пишут код на своих ветках
  - ветки сливаются в master
- GitHub – платформа для размещения своего кода
  - Проекты с открытым исходным кодом (open source) размещаются на GitHub
  - https://github.com/katevi/golang-course/network :)
- Еще есть GitLab, BitBucket, …

# Git: создание веток

- зафиксировать изменения – коммит
  - у каждого коммита есть уникальный хеш
  - последний коммит на бранче – HEAD
  - при сливании веток разработчиков в master коммиты из веток попадут в master
  - можно откатываться на N прошлых коммитов : git reset –hard HEAD~N
    - или на конкретный commit hash: git reset –hard <commit-hash>



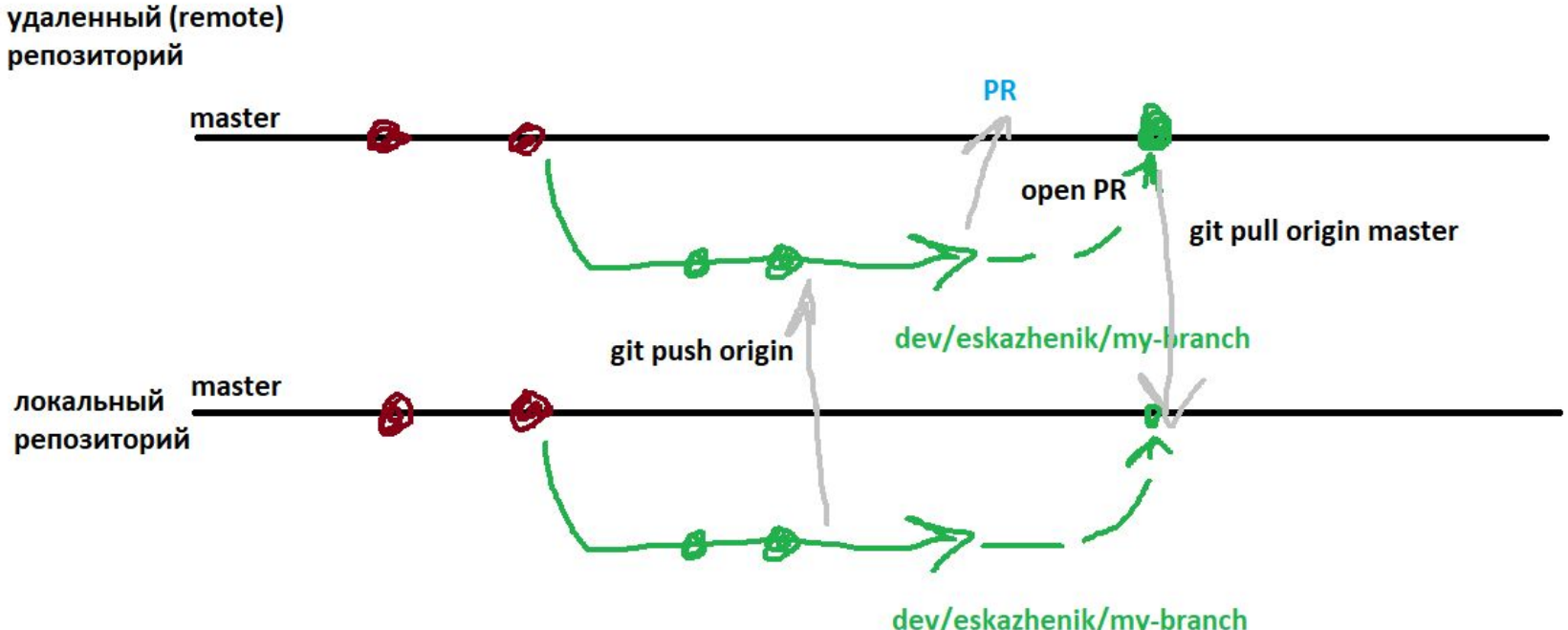
# Git: изменения в репозитории

- git status – показать текущий статус перед коммитом
- https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
- git branch – текущая ветка
- git commit – закоммитить изменения

```
PS C:\Users\katevi\Desktop\Golang\golang-course> git status
On branch dev/eskazhenik/add-linter
Your branch is up to date with 'origin/dev/eskazhenik/add-linter'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   lecture2/homework.md
        modified:   lecture2/links.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        images.png
        lecture2/gophers.png

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\katevi\Desktop\Golang\golang-course> git branch
* dev/eskazhenik/add-linter
  dev/eskazhenik/hw-links-lecture-2
  dev/eskazhenik/layout
  dev/eskazhenik/lecture1-materials
  katevi-add-ci-1
  master
PS C:\Users\katevi\Desktop\Golang\golang-course>
```

# Git: работа с удаленными репозиториями

- git clone – склонировать удаленный репозиторий себе локально
- git push – запушить в удаленный репозиторий, git pull - спуллить



# Ссылки

- Раздел про concurrency: https://go.dev/tour/concurrency/1
- Раздел про concurrency: https://gobyexample.com/goroutines
- Презентация от создателя языка про паттерны: https://go.dev/talks/2012/concurrency.slide#1
- Еще немного про concurrency паттерны: https://go.dev/blog/pipelines
- Git: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository