

Программист - это звучит гордо

Знаниями нужно делиться, иначе они протухают.

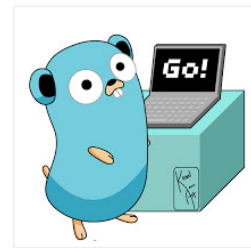
Главная страница

Проекты

вторник, 7 июня 2016 г.

Emacs как IDE для Go

Цель статьи дать обзор инструментов в Emacs для работы с Go кодом. Настроить горячие клавиши, возможно добавить алиасы и сделать их удобнее для повседневного использования - на вашей совести. Хочу отметить, что поддержка языка в Emacs - на высоком уровне: подсветка кода, автодополнение, сниппеты, рефакторинг, подсветка ошибок, отображение документации, тестирование, компиляция и многое другое. Сразу оговорюсь, что проверял я только под Linux, под альтернативные OS могут быть особенности, которые тут не освещены.



Общая настройка Go

Надеюсь у вас уже стоит go, правильно настроены "GOPATH" и т.п., поскольку отдельные плагины чувствительны к подобного рода вещам. Не забудьте добавить "\$GOPATH/bin" в PATH, что бы утилиты, который будем ставить, запускались без указания полного пути. Так же я рассчитываю, что с основами Emacs и Go вы знакомы.

Вся настройка в одном месте

Сначала коротко о настройке. Устанавливаем набор приложений, которые необходимы плагинам:

```
go get -u github.com/nsf/gocode
go get -u github.com/rogppe/godef
go get -u github.com/jstemmer/gotags
go get -u github.com/kisielk/errcheck
go get -u golang.org/x/tools/cmd/guru
go get -u github.com/golang/lint/golint
go get -u golang.org/x/tools/cmd/gorename
go get -u golang.org/x/tools/cmd/goimports
sudo go get -u golang.org/x/tools/cmd/godoc
```

Обратите внимание, что "godoc" нуждается в "sudo", т.к. ставятся в системную директорию, остальные будут установлены в локальный "GOPATH". Я бы посоветовал ставить "godoc" последним, что бы он не создавал директорий с правами root. Из плагинов Emacs понадобятся следующие: [go-mode](#), [go-eldoc](#), [company](#), [company-go](#), [yasnipet](#), [go-rename](#), [multi-compile](#), [flycheck](#), [gotest](#), [go-scratch](#), [go-direx](#), [go-guru](#).

- Как альтернативу company и company-go можно использовать [auto-complete](#) и [go-autocomplete](#).
- Flycheck заменяется [flymake-go](#), так же посмотрите на [golint](#), [go-errcheck](#) и [govet](#).
- Вместо go-scratch иногда используют [go-playground](#).

Полная настройка специфичная для go-mode режима выглядит у меня вот так:

```
(require 'company)
(require 'flycheck)
(require 'yasnipet)
(require 'multi-compile)
(require 'go-eldoc)
(require 'company-go)

(add-hook 'before-save-hook 'gofmt-before-save)
(setq-default gofmt-command "goimports")
(add-hook 'go-mode-hook 'go-eldoc-setup)
(add-hook 'go-mode-hook (lambda ( )
```

```

                                (set (make-local-variable 'company-backends) '(company-go))
                                (company-mode)))
(add-hook 'go-mode-hook 'yas-minor-mode)
(add-hook 'go-mode-hook 'flycheck-mode)
(setq multi-compile-alist '(
  (go-mode . (
    ("go-build" "go build -v"
      (locate-dominating-file buffer-file-name ".git")))
    ("go-build-and-run" "go build -v && echo 'build finish' && eval ./${PWD##*/}"
      (multi-compile-locate-file-dir ".git"))))
  ))

```

Go mode

Теперь по порядку. [Go-mode](#) - базовый пакет, вокруг которого строится остальное. Кроме подсветки, он приносит с собой поддержку команд:

- M-x godef-jump (C-c C-j) - перейти к реализации функции под курсором (вернуться назад, можно через M-*)
- M-x godef-jump-other-window (C-x 4 C-c C-j) - аналогично "godef-jump" только открывается в новом окне
- M-x godoc-at-point - покажет документацию по команде под курсором
- M-x go-goto-imports (C-c C-f i) - перейти к секции "import" текущего файла
- M-x go-goto-function (C-c C-f f) - перейти к началу функции, внутри которой находится курсор
- M-x go-goto-arguments (C-c C-f a) - перейти к аргументам текущей функции
- M-x go-goto-docstring (C-c C-f d) - перейти к комментариям функции
- M-x go-goto-return-values (C-c C-f r) - перейти к описанию возвращаемого значения для функции
- M-x beginning-of-defun (C-M-a) - перейти к началу функции
- M-x end-of-defun (C-M-e) - перейти к концу функции
- Есть базовая поддержка `imenu` для функций и типов

Форматирование исходников я делаю через ["goimports"](#), который установили выше. Он помимо собственно форматирования, умеет добавлять необходимые импорты в текущем файле и вычищать неиспользуемые. Удобно вызывать его автоматически, при сохранении:

```

(add-hook 'before-save-hook 'gofmt-before-save)
(setq-default gofmt-command "goimports")

```

Вручную вызывается через "M-x gofmt".

Вот как это выглядит:

Go eldoc

Плагин [go-eldoc](#) - умеет показывать в строке состояния информацию о переменной или аргументе\возвращаемом значении функции находящейся под курсором. Фактически документация по сигнатурам. Скриншот я нагло украл у автора:

```

package main

import (
    "fmt"
    "strings"
)

func main() {
    s := "qwe"
    s = strings.TrimSpace(s)
    fmt.Println(s)
}

```

1 [article]main.go<article> Go ivy (S) ElDoc Compiling YAS FlyC cmp Wrap

```

0- go-eldoc.el 1+ log.go 2 *scratch*
var ok bool
_, file, line, ok = runtime.Caller(calldepth)
if !ok {
    file = "???"
    line = 0
}
l.mu.Lock()
}
l.buf = l.buf[:0]
l.formatHeader(&l.buf, now, file, line)
l.buf = append(l.buf, s...)
if len(s) > 0 && s[len(s)-1] != '\n' {
    l.buf = append(l.buf, '\n')
}
_, err := l.out.Write(l.buf)
return err
}

-:%%- log.go 51% (148,36) [Not Repo] [[[Go FlyC ap Ys View
formatHeader: (buf *[]byte, t time.Time, file string, line int)

```

Для настройки добавьте вот такие строки:

```

(require 'go-eldoc)
(add-hook 'go-mode-hook 'go-eldoc-setup)

```

Автодополнение

Я предпочитаю для автодополнения [company](#), поэтому ставим ещё [company-go](#) и добавляем в конфиг:

```
(require 'company)
(require 'company-go)
(add-hook 'go-mode-hook (lambda ()
                          (set (make-local-variable 'company-backends) '(company-go))
                          (company-mode))))
```

Это единственная специфичная настройка для Go, сам "company", кто пользуется им, настраивать умеют.



Если вам нравится [auto-complete](#), то для go понадобится [go-autocomplete](#), а настройка описана [здесь](#).

Сниппеты

Тут стандарт - [yasnipet](#), который поставляется с поддержкой Go, если у вас "yasnipet" не включён глобально, достаточно добавить хук для go-mode:

```
(require 'yasnipet)
(yas-reload-all)
(add-hook 'go-mode-hook 'yas-minor-mode)
```

Доступные сниппеты смотрите в [репозитории](#). Если вас они не удовлетворяют, существуют альтернативные наборы: [yasnipet-go](#) и [go-snippets](#), но там, на мой вкус, ничего интересного.


Переименование функций и структур

Переименование самый используемый из методов рефакторинга. Он поддерживается плагином [go-rename](#) и вызывается при помощи "М-х go-rename". Из недостатков - go-rename не работает, если в проекте есть синтаксические ошибки.

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := "qwe"
    s = strings.TrimSpace(s)
    fmt.Println(s)
}
```


 [article]main.go Go ivy (S) ElDoc Compiling YAS FlyC cmp Wrap
Wrote /home/rean/projects/temp/go/article/main.go

```
package main

import "fmt"

func printStr(s string) {
    fmt.Println(s)
}

func main() {
    printStr("qwe")
}
```

 [article]main.go<ReanGD> Go ivy (S) ElDoc Compiling YAS FlyC cmp Wrap
printStr: func((s))

Подсветка ошибок

Сильно ускоряет разработку - подсветка ошибок до компиляции. Для этого ставим [flycheck](#) и делаем общие настройки, по необходимости. Затем переключаемся в буфер в котором включен режим `go-mode` и проверяем, что flycheck видит все необходимые утилиты: "M-x flycheck-verify-setup", у меня получилось так:

```

go-gofmt
- predicate: t
- executable: Found at /usr/bin/gofmt
go-golint
- predicate: t
- executable: Found at /home/user/go/bin/golint
go-vet
- predicate: t
- executable: Found at /usr/bin/go
go-build
- predicate: t
- executable: Found at /usr/bin/go
go-test
- predicate: nil
- executable: Found at /usr/bin/go
go-errcheck
- predicate: t
- executable: Found at /home/user/go/bin/errcheck
Flycheck Mode is enabled.

```

Если что-то не найдено, то вы поставили не все утилиты из списка в начале статьи или не добавили их в PATH. go-test тут не включён, т.к. он срабатывает только для файлов имя которых оканчивается на "_test.go". Настройка заключается в том, что бы включить "flycheck-mode" для режима go:

```

(require 'flycheck)
(add-hook 'go-mode-hook 'flycheck-mode)

```

Об общих настройках flycheck, я надеюсь рассказывать не нужно.

Line	Col	Level	ID	Message (Checker)
				<pre> package main import "fmt" func main() { s := "qwe" fmt.Println(s) } </pre>
1		Error	R*Flycheck errors*	for buffer main.go<ReanGD> Flycheck e
2		Info	[article]	main.go<ReanGD> Go ivy (S) ElDoc Comp

undefined: s

Поклонники flymake - сделают аналогичное при помощи пакета [flymake-go](#) (инструкции по настройке у [автора в репозитории](#)), а так же [goflymake](#). При желании посмотрите на [golint](#), [go-errcheck](#) и [govet](#). Но на мой взгляд flycheck функциональнее.

Компиляция и запуск

Тут сразу две проблемы:

- Стандартный модуль [compile](#) позволяет настроить только одну команду для режима. А я хочу меню с компиляцией, компиляцией и запуском, запуском тестов и т.п.

- Я перерыл пол интернета и не нашёл способ обнаружения корня проекта на Go. Если у вас сложная иерархия проекта и открыт файл внутри этого проекта - найти место где нужно грубо говоря запустить build это та ещё задача.

Первая проблема решается при помощи плагина [multi-compile](#), подробнее читайте в [этой статье](#). А вот вторая в каждом конкретном случае решается особо. Для себя я решил считать корнем проекта ту директорию, где лежит ".git", поскольку любой проект начинаю с "git init". А для экспериментов с языком - использую go-scratch.

Настройка для плагина multi-compile выглядит вот так:

```
(require 'multi-compile)
(setq multi-compile-alist '(
  (go-mode . (
    ("go-build" "go build -v"
      (locate-dominating-file buffer-file-name ".git"))
    ("go-build-and-run" "go build -v && echo 'build finish' && eval ./${PWD##*/}"
      (multi-compile-locate-file-dir ".git"))))
  ))
```

Да, на первый взгляд - сложно, но если разобраться, то не очень.

Тут написано что для файла открытого в режиме "go-mode" добавить два пункта меню "go-build" и "go-build-and-run". Первый вызывает консольную команду "go build -v". Второй "go build -v && echo 'build finish' && eval ./\${PWD##*/}".

Как рабочая директория у обеих команд устанавливается та, внутри которой лежит ".git", т.е. если у нас открыт файл "~/go/go1/go.go" плагин поищет директорию ".git" внутри "~/go/go1/", если не найдёт, ищет уровнем выше - в "~/go/". Как только нужная директория найдена - она считается рабочей из которой и вызываются скрипты.

Теперь о страшных bash командах: "go build -v" вопросов не вызовет - стандартная компиляция проекта. А вот вторая сложнее:

```
go build -v && echo 'build finish' && eval ./${PWD##*/}
```

Конструкция "\${PWD##*/}" разворачивается в имя последнего элемента текущей директории. Н-р для "~/go/super_project", получим "super_project". Т.е. для этого примера, команда превращается вот в это:

```
go build -v && echo 'build finish' && eval ./super_project
```

Теперь читается проще - компилируем, пишем в консоль, что компиляция завершилась и запускаем получившийся бинарник. Go по умолчанию называет бинарник по имени директории, где находится проект, в нашем случае это "super_project".

Меню с командами компиляции вызывается вот так: M-x multi-compile.

Больше примеров настройки на [github](#).

Тестирование

Я для тестирования пользуюсь [goconvey](#), который умеет самостоятельно обнаруживать изменения в файлах, и перезапускать тесты. Но часто запуск всего набора тестов - лишнее действие, которое сильно грузит процессор и заставляет меня менять контекст переключаясь в браузер. Поэтому хочется иметь возможность запустить только один тест или все тесты в данном файле оставаясь в Emacs. В этом нам поможет пакет [gotest](#).

Дополнительной настройки он не требует (ну может только горячие клавиши назначить). Умеет он следующее:

- M-x go-test-current-test - запустить тест внутри которого находится курсор
- M-x go-test-current-file - запустить тесты внутри текущего файла
- M-x go-test-current-project - запустить тесты для текущего проекта
- M-x go-test-current-benchmark - по аналогии - запустить бенчмарк внутри которого находится курсор
- M-x go-test-current-file-benchmarks - запустить бенчмарки в файле
- M-x go-test-current-project-benchmarks - запустить бенчмарки в проекте

Локальный Go Playground

Что-бы попробовать как работает та или иная часть Go, обычно пользуются [play.golang.org](#), но у него большой недостаток - он работает в браузере, это не удобно. А локально создавать проект для каждой мелочи лень. Эту нишу покрывает плагин [go-scratch](#). После команды "M-x go-scratch" откроется новый буфер с заготовкой функции main. Компиляция запускается сочетанием клавиш "C-c C-c".

Существует похожий плагин [go-playground](#), но мне он не понравился, т.к. физически создаёт файл на диске, а чистить за ним не

```
package main

import "fmt"

func main() {
    s := "Hello go"
    fmt.Println(s)
}
```

1 [article]main.go<ReanGD> Go ivy (S) EDoc Compiling YAS FlyC cmp Wrap

хочется. (в [комментариях](#) автор плагина объясняет почему так сделано и как с этим жить)

Отображение структуры кода

Если вас не устраивает `imenu`, поддержку которого добавляет `go-mode`, то воспользуйтесь [go-direx](#), который, нужно отдать должное, покажет структуру текущего кода в гораздо более наглядном виде. Как-то настраивать его не нужно, достаточно вызвать "M-x go-direx-switch-to-buffer".

```
package crawler

import (
    "errors"
    "net/url"
    "strings"

    "golang.org/x/net/html"
    "golang.org/x/net/html/atom"
)

var (
    // ErrDataExtractorUnexpectedNodeType - found unexpected node type
    ErrDataExtractorUnexpectedNodeType = errors.New("data_extractor.dataExtractor.parseNode: unexpected node ty
)

// HTMLMetadata extracted meta data from HTML
type HTMLMetadata struct {
    // [URL]hostname
    URLs map[string]string
    // [URL]error
    WrongURLs map[string]string
    Title      string
    MetaTagIndex bool
}

type dataExtractor struct {
    meta      *HTMLMetadata
    baseURL   *url.URL
}

1 [go-web-search]crawler/data_extractor.go Go ivy (S) EDoc Compiling YAS FlyC cmp Wrap :master
No window numbered 2
```


Go guru

Я не могу не упомянуть ещё об одном плагине - [go-guru](#). Он умеет много интересных и полезных вещей, но в рамки статьи они не поместятся. Подробно почитать о них можно в [oracle-user-manual](#), там правда речь идёт о предыдущем названии "go-oracle", сейчас проект переименован в "go-guru", но суть осталась прежней.

Для затравки покажу как найти места откуда вызывается функция. Вызываем:

```
M-x go-guru-set-scope
```

И вводим пакет с которым работает (в моём случае это "github.com/ReanGD/go-web-search"). После этого ставим курсор на интересующую функцию и вызываем

```
M-x go-guru-callers
```

В отдельном буфере отобразится список мест откуда функция вызывается. На gif надеюсь понятнее:

```
return ""
}

func (extractor *dataExtractor) getAttrValLower(node *html.Node, attrName string) string {
    return strings.ToLower(extractor.getAttrVal(node, attrName))
}

func (extractor *dataExtractor) isEnabledLinkParse() bool {
    return extractor.metaTagFollow && extractor.noIndexLvl == 0
}

func (extractor *dataExtractor) processLink(link string) {
    if link == "" {
        return
    }
    relative, err := url.Parse(strings.TrimSpace(link))
    if err != nil {
        extractor.meta.WrongURLs[link] = err.Error()
        return
    }

    parsed := extractor.baseURL.ResolveReference(relative)
    urlStr := NormalizeURL(parsed)
    parsed, err = url.Parse(urlStr)

    if (parsed.Scheme == "http" || parsed.Scheme == "https") && urlStr != extractor.baseURL.String() {
        extractor.meta.URLs[urlStr] = NormalizeHostName(parsed.Host)
    }
}

[go-web-search]crawler/data_extractor.go Go ivy (S) ELDoc Compiling YAS FlyC cmp Wrap -master
Quit
```

Из недостатков - работает медленно, но в любом случае - руками делать подобные вещи дольше. И как я слышал, авторы сейчас серьёзно взялись за проект, надеюсь работать станет удобнее и быстрее.

Автор: [ReanGD](#) на [10:53](#) 4 Comments

Ярлыки: [emacs](#), [go](#), [ide](#)



Присоединиться к обсуждению...

ВОЙТИ С ПОМОЩЬЮ

ИЛИ ЧЕРЕЗ DISQUS ?

Имя

**Alexander I. Grafov** • 2 года назад

Спасибо за статью! Как автор go-playground скажу несколько слов в его защиту :) В нем можно удалять сниппеты через M-x go-playground-remove-current-snippet (забиндив куда хочется) — удаляет папку вместе с файлами. Сохранение сниппетов в файлы под GOPATH — это фишка, т.к. можно использовать все утилиты и расширения Emacs для работы с go, которые в основном хотят файл на входе. Поэтому все что доступно при настройке Emacs для golang, доступно в go-playground. Опять же локальная библиотека сниппетов.

^ | ▾ • Ответить • Поделиться ▸

**ReanGD** Модератор ➔ Alexander I. Grafov • 2 года назад

Попробую на досуге сравнить насколько go-scratch хуже работает с расширениями, может и правда с go-playground удобнее будет)

А пока дал ссылку на ваш комментарий в статье.

1 ^ | ▾ • Ответить • Поделиться ▸

**corvinusz** • 2 года назад

Для Go есть много хороших линтеров. Для установки их скопом можно поставить gometalinter

```
$ go get -u github.com/alecthomas/gometalinter...
```

и выполнить установку линтеров

```
$ gometalinter --install --update.
```

А в emacs все это интегрируется как пакет 'flycheck-gometalinter'. Я использую 'use-package' (можно обойтись и 'require' - см. документацию по 'flycheck-gometalinter')

```
;;gometalinter
(use-package flycheck-gometalinter
:ensure t
:config
(progn
(flycheck-gometalinter-setup))
)
;; gometalinter: skips 'vendor' directories and sets GO15VENDOREXPERIMENT=1
(setq flycheck-gometalinter-vendor t)
;; gometalinter: only enable selected linters
(setq flycheck-gometalinter-disable-all t)
(setq flycheck-gometalinter-enable-linters '("golint" "vet" "vetshadow" "golint" "ineffassign" "goconst" "gocyclo"
"errcheck" "deadcode"))
```

1 ^ | ▾ • Ответить • Поделиться ▸

**ReanGD** Модератор ➔ corvinusz • 2 года назад

Спасибо за дельный комментарий, как-то упустил я такую классную штуку. Обязательно попробую.

^ | ▾ • Ответить • Поделиться ▸

