Go进阶训练营
第4课
Go 工程化实践答疑

大明

目录

工程项目结构

1. 实践：经济适用版项目布局
    1. 单体应用
    2. 单体引用切分
    3. 切分原则
2. 作业讲解
    1. 项目结构
    2. protobuf 入门
    3. Wire 入门

经济适用版项目布局——单体应用

可以考虑替换的：

1. data 里面放的是PO，也可以改名叫做 model

2. 在业务逻辑不复杂的情况下，DO 和 PO 可以只保留一个。也就是，在这种场景之下，也可以改名叫做 model。它会被直接用于持久化，以及承担轻量的业务逻辑——如果要承担很多逻辑，最好将 DO 分离出来

3. pkg 里面放着各种通用的、与业务无关的代码

4. web 直接暴露 HTTP 接口。它主要调用 biz 的方法来完成业务逻辑，而后将数据转换成 VO 暴露出去。VO 是建议大家保留的，因为页面的需求是千变万化的，但是 model 之类的是很稳定的

5. web 和 task 都依赖于 biz。task 提供一些定时或者周期任务

6. cmd 也尽量依赖于 biz，相当于只是将业务逻辑暴露为简单的命令行

7. pkg 里面绝对不能依赖任何别的包

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        - PO
        user_repo.go
        user_cache.go
    - biz
        - DO
        user.go
    - pkg
        - strings
            utils.go
    - task
    - web
        - vo
        user.go
```

经济适用版项目布局——单体应用

如果单体应用内容比较多：

1. web, biz, data, task 都进一步按照业务进一步划分；

2. 不必每一部分都直接全部分到最细，比如说 data 里面 user 部分很多内容，就单独一个文件夹放着，但是其它部分不多，就直接丢到 data下，将来再考虑拆分

3. 在按照业务细分之后，可以考虑使用集中的 VO 目录，也可以直接定义在各自的业务文件夹下。例如 user 的 vo 可以直接放在 /web/user里面，也可以有一个 /web/vo。biz 和 data 也是类似处理；

```
22
23  -cmd
24  -api
25  -configs
26  -go.mod
27  -main.go
28  -internal
29      - data
30          -user
31          order.go
32      - biz
33          -user
34            user.go
35            buyer.go
36          -order
37          -product
38      - pkg
39          - strings
40              utils.go
41      - task
42      - web
43          -user
44          -order
45          -product
46
```

经济适用版项目布局——拆分单体应用

结合我们第一课的内容：

1. 如果目录结构已经演进到了这个地步，那么只需要将某个业务的全部层级里面的代码拆出来，挪到一个新的项目就可以

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        buyer.go
    - biz
        user.go
        buyer.go
    - pkg
        - strings
            utils.go
    - task
    - web
        user.go
```

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        order.go
    - biz
        order.go
    - pkg
        -strings
            utils.go
    - task
    - web
        order.go
```

```
-go.mod
-pkg
 -strings
   utils.go
```

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        -user
        order.go
    - biz
        -user
          user.go
          buyer.go
        -order
        -product
    - pkg
        - strings
            utils.go
    - task
    - web
        -user
        -order
        -product
```

经济适用版项目布局——拆分单体应用

结合我们第一课的内容：

1. 拆分出一个 web 应用和多个领域服务，以及可能的一个 kit

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        user.go
    - biz
        user.go
    - pkg
        -strings
            utils.go
    - task
    - service
        user.go
```

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        order.go
    - biz
        order.go
    - pkg
        -strings
            utils.go
    - task
    - service
        order.go
```

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - pkg
        -strings
            utils.go
    - web
        order.go
        user.go
```

```
-go.mod
-pkg
 -strings
  utils.go
```

```
-cmd
-api
-configs
-go.mod
-main.go
-internal
    - data
        -user
        order.go
    - biz
        -user
          user.go
          buyer.go
        -order
        -product
    - pkg
        - strings
            utils.go
    - task
    - web
        -user
        -order
        -product
```

各种布局的划分思路

核心原则：高内聚，低耦合

1. 横向按照层级分，或者说按照功能分：如 service, biz, data。这种划分有很明显的层级结构

2. 竖向按照业务分

问题：什么时候横向分？什么时候竖向分？

1. 因为业务复杂度演进而带来的，应该竖向分。典型例子是针对商家还是买家，细化用户服务

2. 因为引入中间层级，尝试维系“内聚耦合”的，应该横向分。典型例子是在 data 里面引入 cache

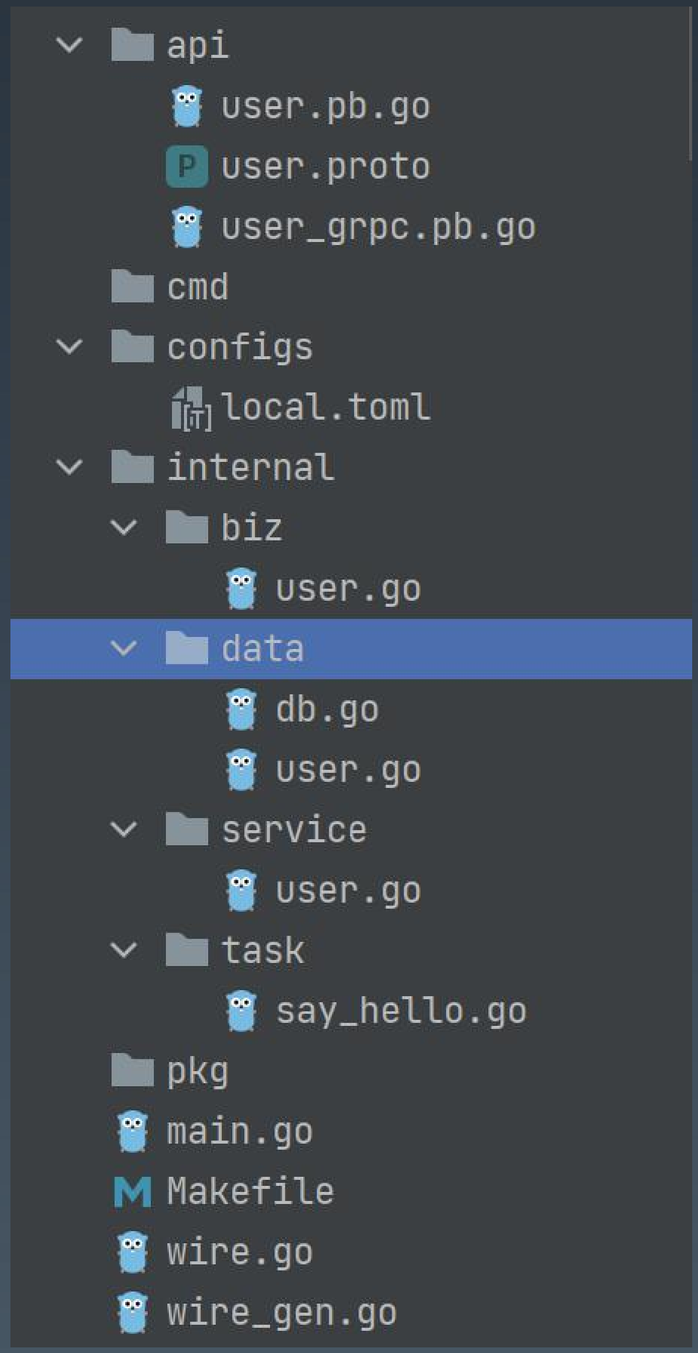
```
22
23  -cmd
24  -api
25  -configs
26  -go.mod
27  -main.go
28  -internal
29      - data
30          -user
31          order.go
32      - biz
33          -user
34            user.go
35            buyer.go
36          -order
37          -product
38      - pkg
39          - strings
40              utils.go
41      - task
42      - web
43          -user
44          -order
45          -product
46
```

项目布局一些简单粗暴的原则

1. 需要被别的项目使用的代码，丢过去pkg。要非常谨慎，因为 pkg 的代码丢过去就收不回来（类似于，一个大写开头的方法，暴露出去了就收不回来了）
2. 纯粹的微服务项目，web 项目，没啥代码在项目间复用的代码，全部丢过去 internal
3. 额外的命令，比如说一些工具类命令，一些修复数据的命令，丢过去 cmd
4. 如果是中间件，除了是用户能用的接口、结构体，其它都丢过去 internal
5. 以公司规范为准，没有规范就以这个课程的 v2 版本为主。如果项目特别小，就是 CRUD，可以裁剪一部分 v2。

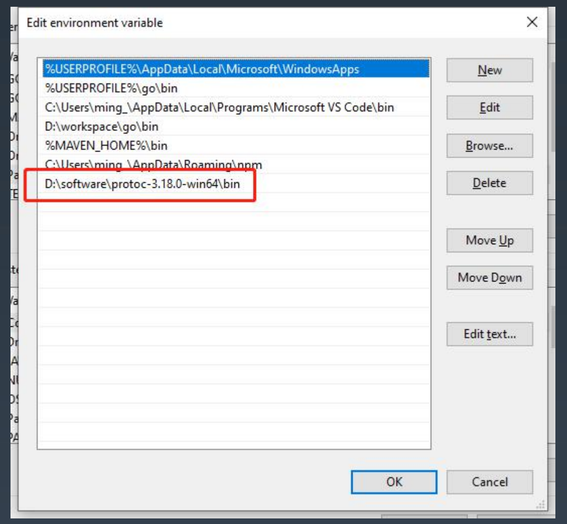
作业讲解 —— 项目结构

1. 根据自己对课程的理解，主要是参考 v2 来构建一个简单的项目
2. api 目录，可以单独目录，也已进一步划分（适用于复杂项目）。最好是单独一个 api 的项目（参考 API 设计一节）。api 最好是使用一些比较公认的 API 定义方式，比如说 protobuf
3. 引入 wire。wire 的入门稍微有点繁琐



作业讲解 —— protobuf 入门

1. 安装 protoc 环境，https://grpc.io/docs/protoc-installation。
2. 以windows 为例。从 https://github.com/protocolbuffers/protobuf/releases/tag/v3.18.0 下载
3. 解压缩将 bin 目录加入到环境变量
4. 安装 golang 和 grpc 的插件
5. 执行 protoc 命令



- **Go plugins** for the protocol compiler:

  1. Install the protocol compiler plugins for Go using the following commands:

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.26
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1
```

  2. Update your `PATH` so that the `protoc` compiler can find the plugins:

```
$ export PATH="$PATH:$(go env GOPATH)/bin"
```

```
$ protoc --go_out=. --go_opt=paths=source_relative \
    --go-grpc_out=. --go-grpc_opt=paths=source_relative \
    helloworld/helloworld.proto
```

Wire

https://blog.golang.org/wire

手撸资源的初始化和关闭是非常繁琐，容易出错的。上面提到我们使用依赖注入的思路 DI，结合 google wire，静态的 go generate 生成静态的代码，可以在很方便诊断和查看，不是在运行时利用 reflection 实现。

```
type Message string

type Greeter struct {
    // ... TBD
}

type Event struct {
    // ... TBD
}
```

```
func main() {
    message := NewMessage()
    greeter := NewGreeter(message)
    event := NewEvent(greeter)

    event.Start()
}
```

```
func NewMessage() Message {
    return Message("Hi there!")
}
```

```
// wire_gen.go

func InitializeEvent() Event {
    message := NewMessage()
    greeter := NewGreeter(message)
    event := NewEvent(greeter)
    return event
}
```

```
// wire.go

func InitializeEvent() Event {
    wire.Build(NewEvent, NewGreeter, NewMessage)
    return Event{}
}
```

```
func NewGreeter(m Message) Greeter {
    return Greeter{Message: m}
}

type Greeter struct {
    Message Message // <- adding a Message field
}
```

目录

API 设计

1. 实践——中间件 API 设计方法论
2. 知识梳理：API errors

中间件 API 设计方法论——大明版

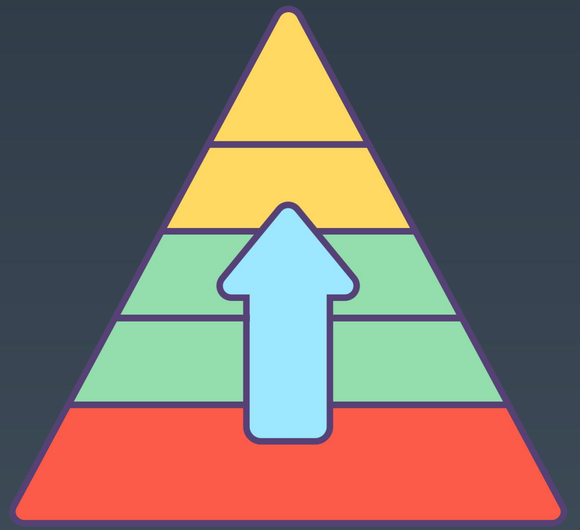
自底向上

自顶向下

中间件 API 设计方法论——自顶向上

1. 一般就是最开始不清楚该怎么设计，所以直接写实现，没有接口设计
2. 在实现过程中，发现某个地方可能存在变更的可能，于是将这个地方抽象出来，做成了一个接口
3. 在不断抽象接口的时候，会发现有一些接口比另外一些接口更加抽象，也就是更为高级，最终组成一个类似金字塔的层级
4. 难点在于识别变更点——依赖于个人见识和经验

我在 web 框架小课就是参考这种思路的



自底向上

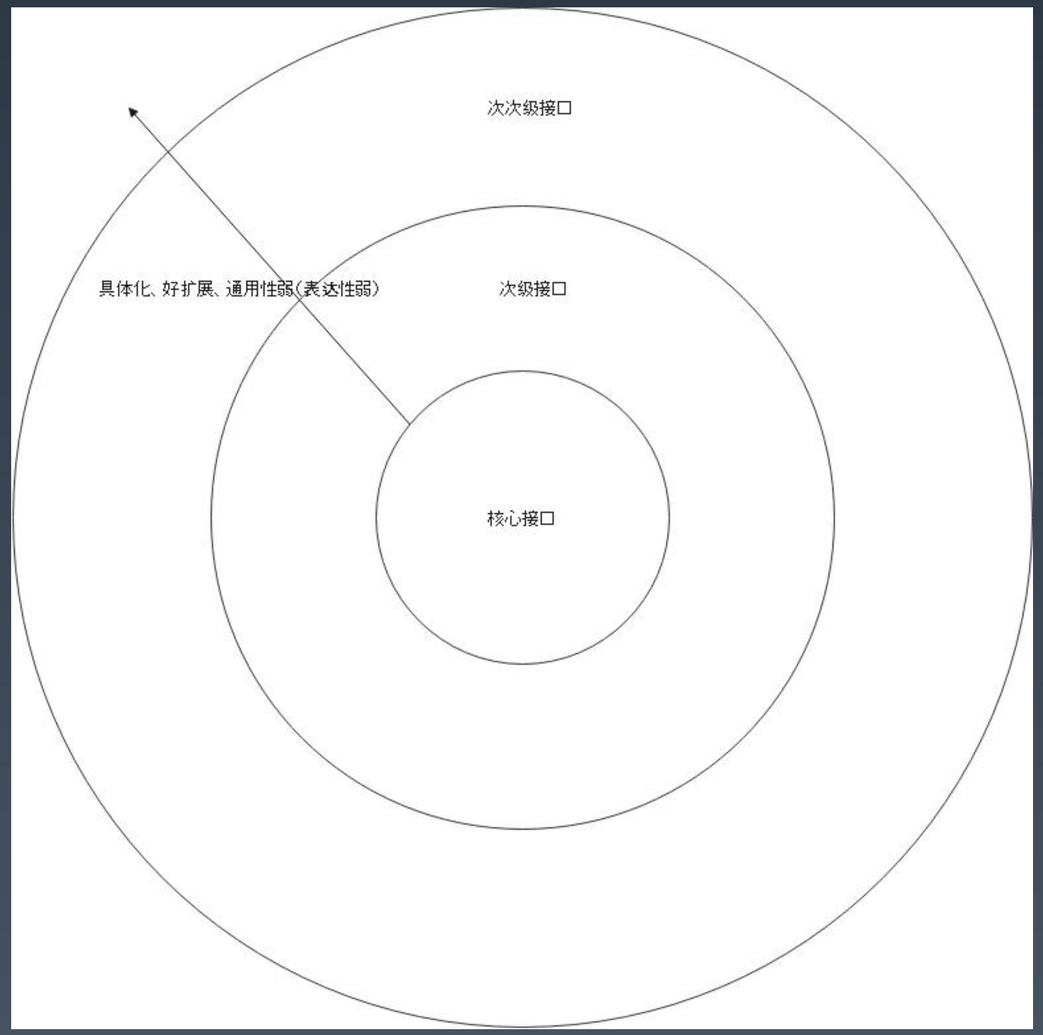
中间件 API 设计方法论——自顶向下

1. 设计核心API —— 一组能够描述清楚自己整个功能的 API
2. 在实现核心 API 的时候，发现某种实现，需要引入新的接口，这些接口就是支撑核心接口的次级接口
3. 围绕着次级接口的实现设计次次级接口

自顶向下

中间件 API 设计方法论——自顶向下

1. 随着接口不断向外扩张，其表达性开始减弱。比如说核心接口表达了整个模块的功能，那么次级接口可能只是表达了模块的某个方面的功能...最外围的接口可能只是这个模块一个非常具体的点可以有不同的做法
2. 同时抽象性下降，愈发具体。接口更加具体带来的好处就是好理解，易扩展。比如说大多数中间件，直接设计出来就是为了用户扩展的接口，普遍都是外围接口，例如各种 Filter 接口，Interceptor 接口
3. 越是靠近核心的接口，开发者越不想你扩展，甚至可能根本没有提供接入自定义实现的手段



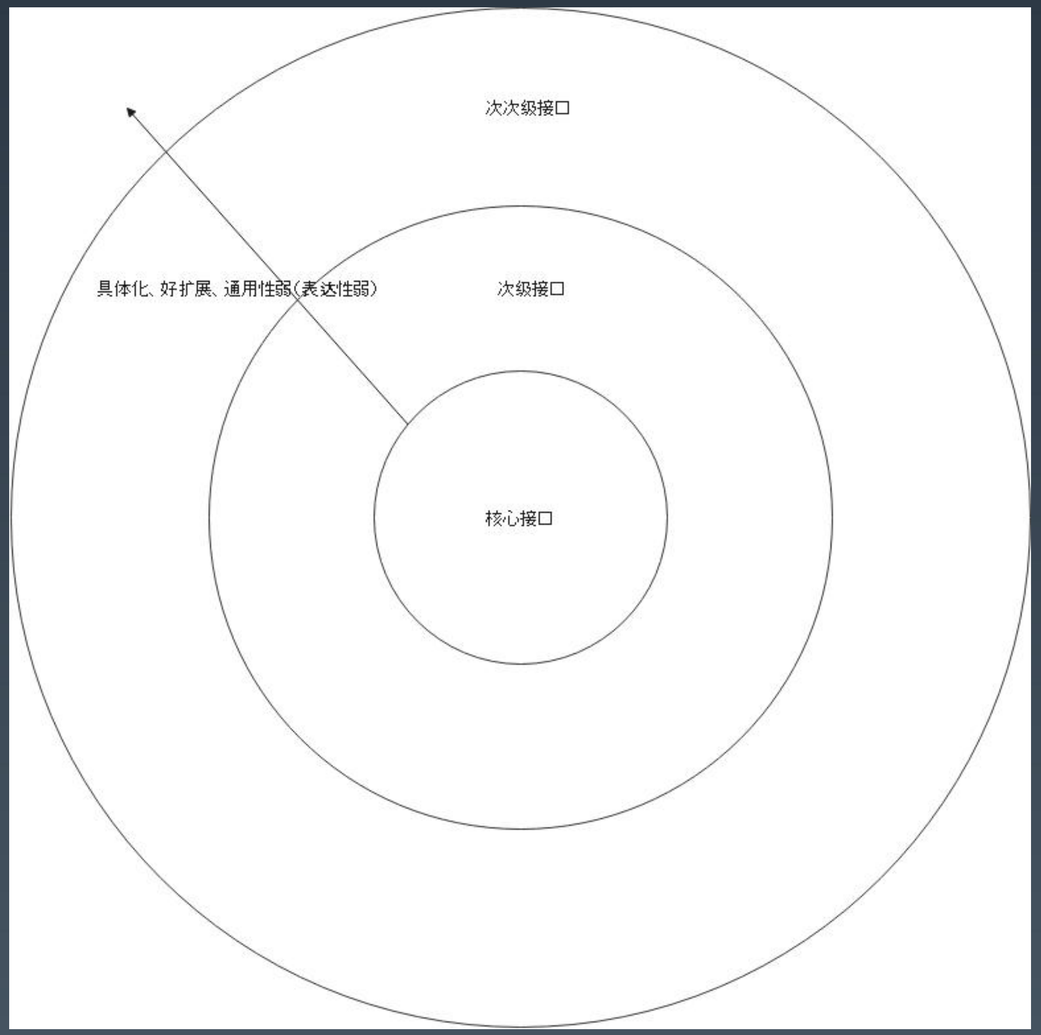
中间件 API 设计方法论——自顶向下的一个例子

1. 核心接口：QueryBuilder。描述了整个模块是干啥的：也就是构建一个查询，返回 SQL 和查询参数。

剩下的不管是实现，还是别的接口，都是服务于这个接口的。

```go
// QueryBuilder is used to build a query
type QueryBuilder interface {
    Build() (*Query, error)
}

// Query represents a query
type Query struct {
    SQL string
    Args []interface{}
}
```



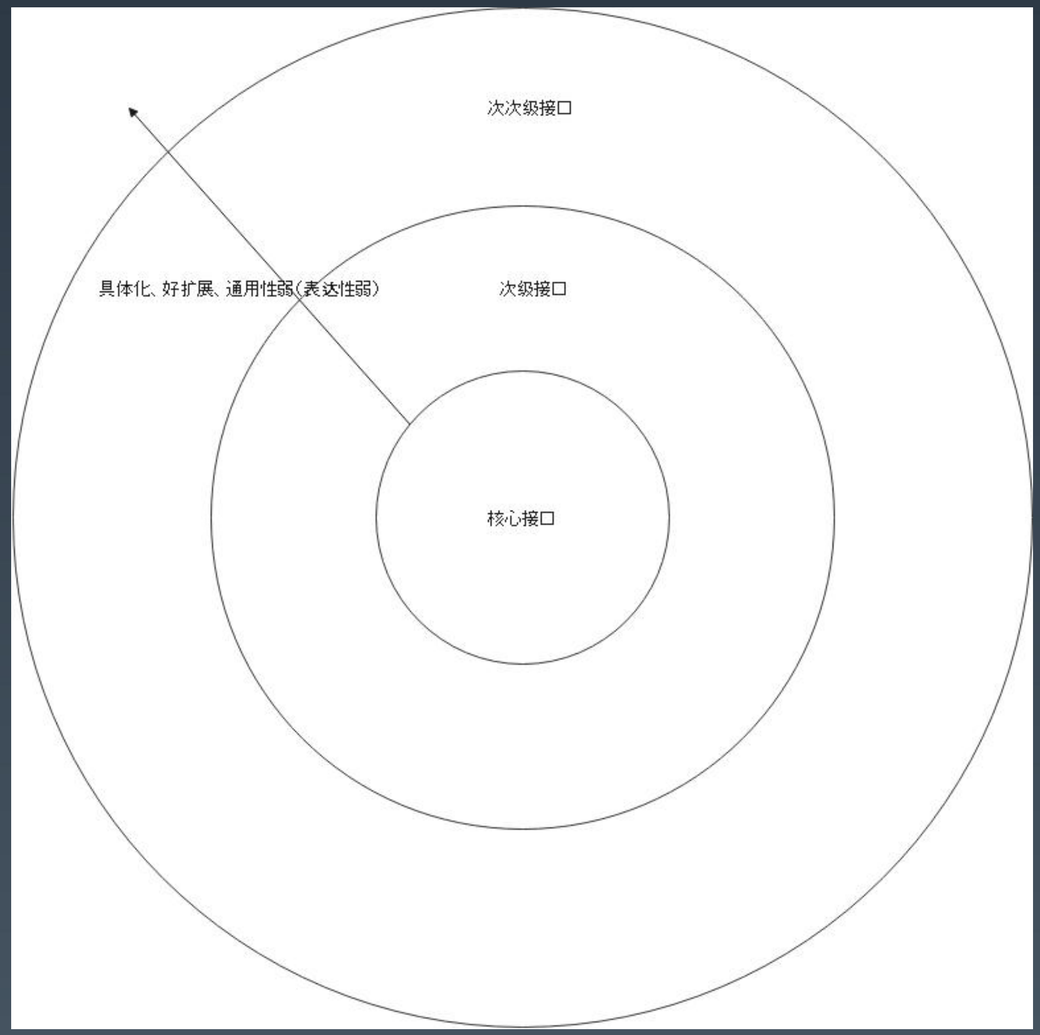
中间件 API 设计方法论——自顶向下的一个例子

1. 核心接口：QueryBuilder。描述了整个模块是干啥的：也就是构建一个查询，返回 SQL 和查询参数。
2. 次级接口：Selector 是 QueryBuilder 的直接实现，为了支撑 Selector 的实现，引入了新的接口

```go
// Select starts a select query. If columns are empty, all
func (*DB) Select(columns...Selectable) *Selector {
    panic( v: "implement me")
}
```

```go
// Where accepts predicates
func (*Selector) Where(predicates...Predicate) *Selector {
    panic( v: "implement me")
}
```

```
Type Selectable implemented in 4 types
Aggregate (in github.com/gotomicro/eql/aggregate.go
Column (in github.com/gotomicro/eql/column.go)
RawExpr (in github.com/gotomicro/eql/expression.go)
columns (in github.com/gotomicro/eql/column.go)
```



API Errors

1. 下游错误码
2. 内部业务错误码
3. 上游可感知错误码

考虑：

1. 你是否希望你的上游知道你的下游的错误？如果不希望，请转化
2. 你的上游能否理解你的下游的错误？如果不能，请做转化
3. 你是否信任你的上游？不信任，请将内部业务错误码转换成安全的错误码（不会暴露过多的细节）

目录

实践：Configs VS Options

1. 配置都是简单的类型，比如说字符串、数字等，优先考虑 Configs，可以直接从文件加载
2. 配置含有复杂结构体，比如说 Repo 的实现，你允许用户是否传入一个 cache ，以缓存数据，cache 实例的构建本身非常复杂，那么就只能使用 Options
3. 你希望能够灵活监听配置变更，基本上只能使用 Configs
4. 你提供了绝大部分配置的默认值——这些配置都是必须的，优先考虑 Options
5. 你觉得没有必要引入一个新的 Configs 结构体，使用 Options。例如 DB 可以直接包含所有的配置，也可以 DB 包含一个 Config 的字段，取决于个人
6. 配置的是行为，使用 Options。比如说，允许用户传入 log 方法

我的原则：遇事不决用 Options

实践：一种不太好但是很好用的实践

1. 用代码来做配置
2. 源自这样的一种理念：既然在不同环境下，你要使用不同的配置文件，比如说`dev.yaml`，`prod.yaml`，我干什么不直接在代码里面写呢？
3. 缺点是需要重新编译部署

```go
// Env 会在编译的时候注入值

// Env go build -ldflags "-X 'main.Env=aaa'"
var Env = "dev"
func main() {
    cfg := InitConfig(Env)
    println(cfg.Name)
}

type Config struct {
    Name string
}
```

```go
func InitConfig(env string) Config {
    if env == "dev" {
        return initDevConfig()
    }
    return initProdConfig()
}

func initDevConfig() Config {
    return Config{
        Name: "This is dev",
    }
}

func initProdConfig() Config {
    return Config{
        Name: "This is prod",
    }
}
```

目录

GOPROXY 访问内网仓库

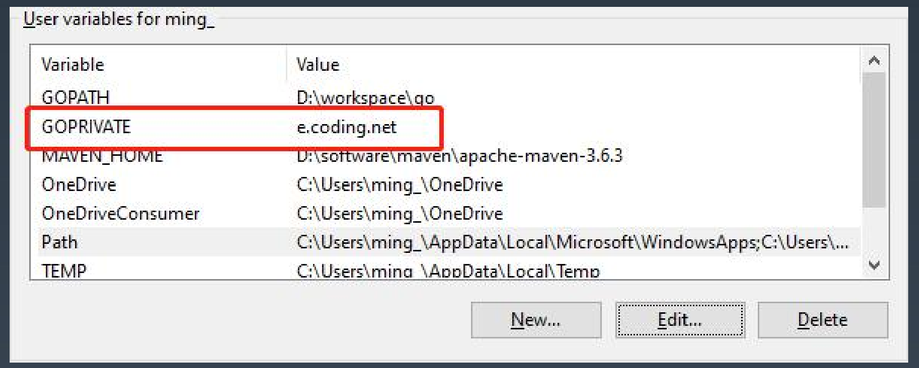
goproxy 配置访问公司内网的 git server：

- 用户本地配置 GONOSUMDB=github.com/private
- goproxy server 配置 exclude 进行排除掉所代理仓库
- goproxy server 配置 SSH Key，并且在仓库添加只读权限
- goproxy server 配置 .gitconfig 把 ssh 替换成 http 方式访问

```
[url "git@github.com:"]
    insteadOf = https://github.com/
[url "git@gitlab.com:"]
    insteadOf = https://gitlab.com/
```

如何使用 GO 私有 repo —— 以 coding 为例

1. 配置 GOPRIVATE
2. 在 coding 上加入自己的 SSH 公钥
3. 配置 ~/.gitconfig




```
[url "e.coding.net:"]
insteadOf = http://e.coding.net/
[url "git@e.coding.net:"]
insteadOf = https://e.coding.net/
```

目录

测试

1. 扩展知识： Go mock 用法
2. 实践：性能测试 - go benchmark 测试
3. 实践：丐中丐版 TDD

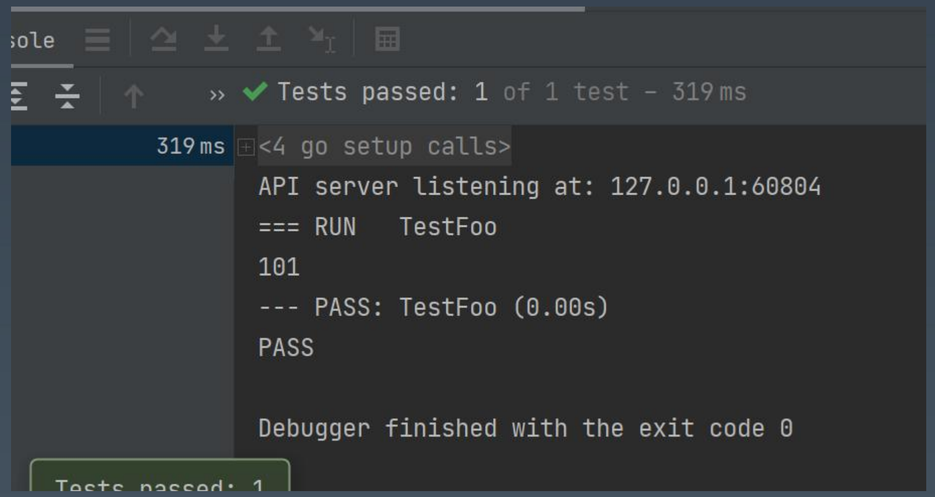
实践：go mock 用法

1. 安装：go install github.com/golang/mock/mockgen@v1.6.0

2. mockgen -source=foo.go -destination=mock_test/foo_mock.go

```go
type Foo interface {
    Bar(x int) int
}

func SUT(f Foo) {
    println(f.Bar( x: 99))
}
```




```go
func TestFoo(t *testing.T) {
    ctrl := gomock.NewController(t)

    // Assert that Bar() is invoked.
    defer ctrl.Finish()

    m := mock_fourth.NewMockFoo(ctrl)

    // Asserts that the first and only call to Bar() is passed 99.
    // Anything else will fail.
    m.EXPECT().
        Bar(gomock.Eq( x: 99)).
        Return( rets...: 101)

    SUT(m)
}
```

实践：编写易于测试的代码

1. 外部传入实例，而不是内部自己创建；如果不行，就使用可以修改的包变量。最好是将 DB, RPC 等访问第三方的做成成员变量，而后在创建的时候传入 mock 的版本
2. 面向接口编程：因为不支持重写，所以只能面向接口编程
3. 可以尝试将复杂逻辑抽取出来单独测试，所有需要的组件都是作为参数传入进去

实践：性能测试 —— benchmark

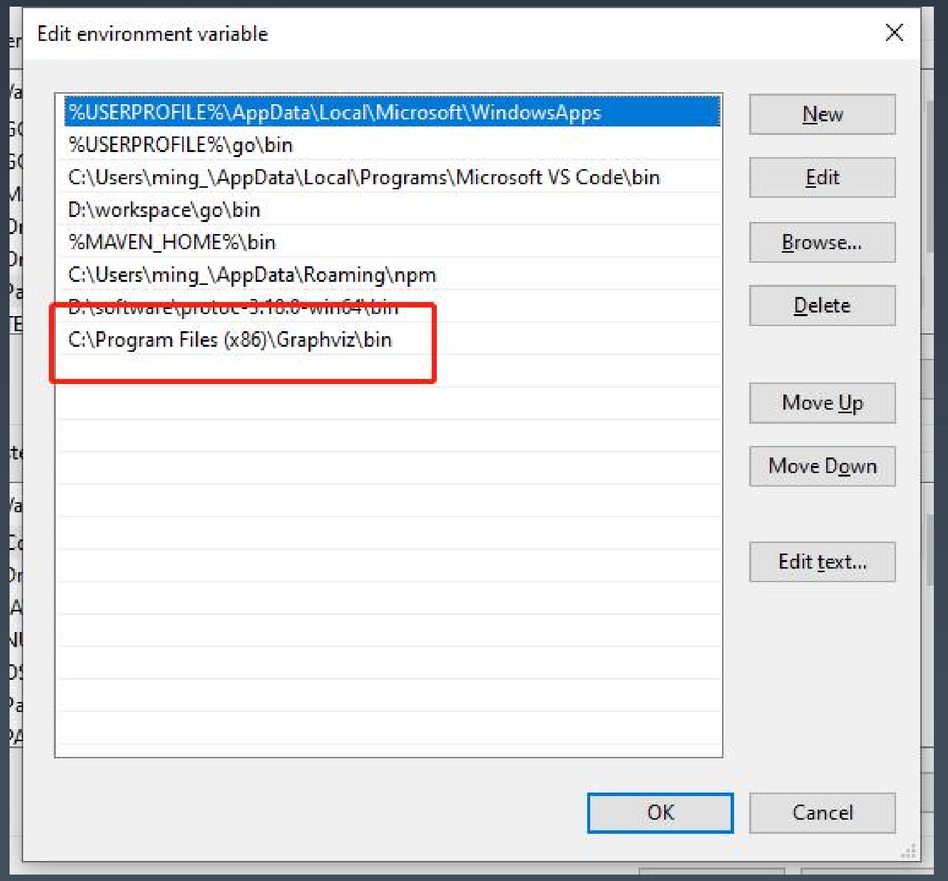
```go
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

```go
func BenchmarkFib(b *testing.B) {
    Fib( n: 40)
}
```

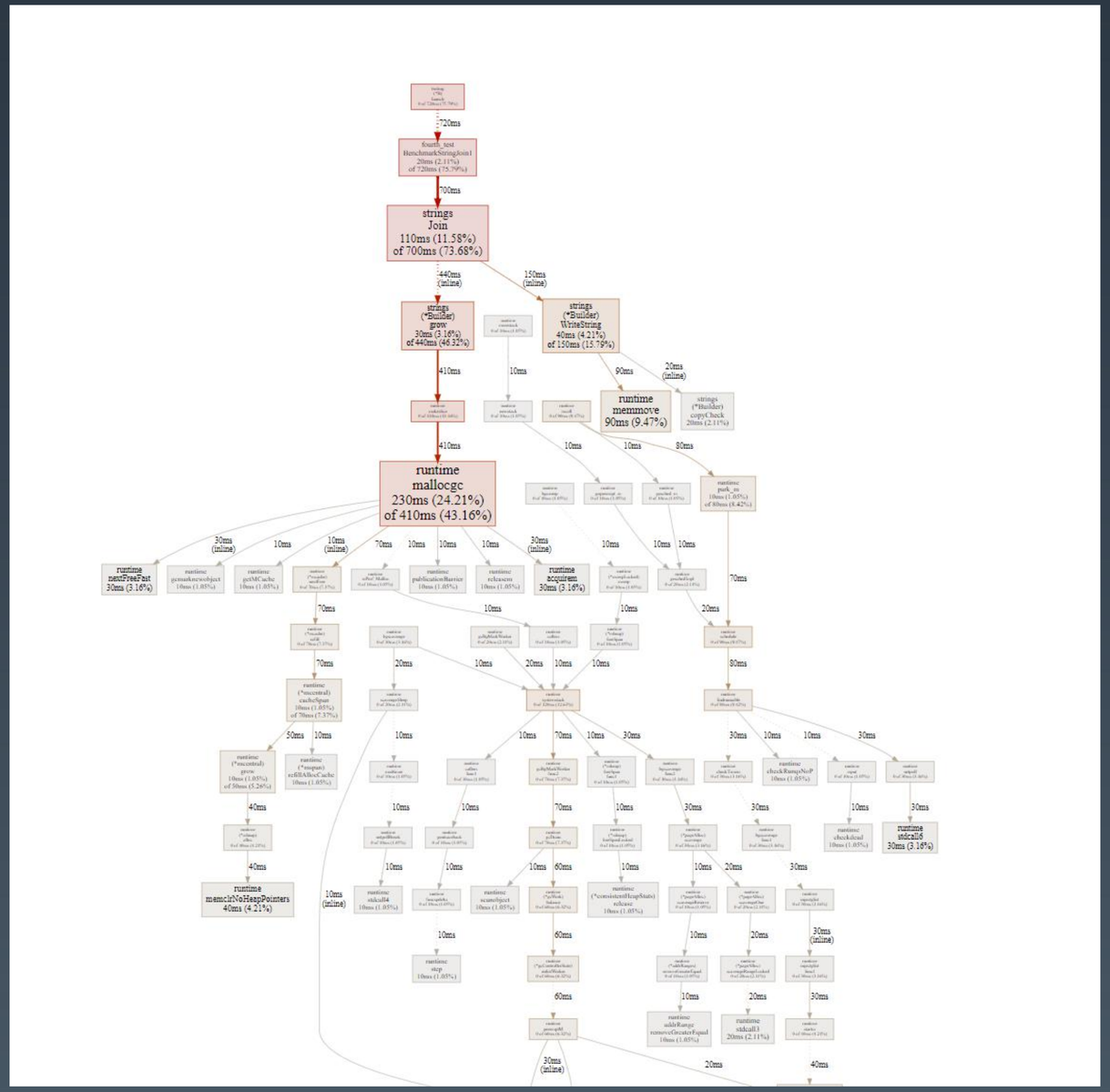
```
PS D:\workspace\go\src\geekbang\geekbang-go-camp\test> go test `-bench=.
goos: windows
goarch: amd64
pkg: geekbang/geekbang-go-camp/test
cpu: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz
BenchmarkFib-12         1000000000               0.4424 ns/op
PASS
ok      geekbang/geekbang-go-camp/test  10.318s
PS D:\workspace\go\src\geekbang\geekbang-go-camp\test>
```

实践：性能测试 —— google pprof 工具

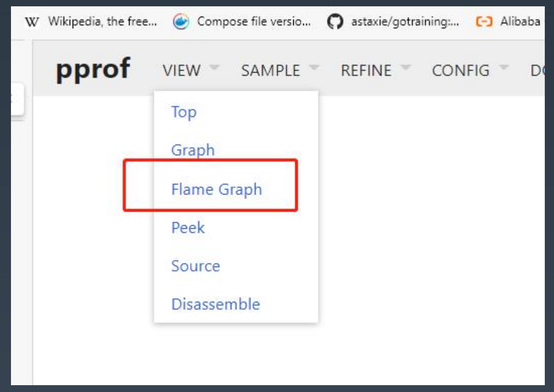
1. -benchmem 等效于`ReportAllocs`
2. 输出 cpuprofile：-cpuprofile cpu.prof
3. 使用pprof 工具来分析：go get -u github.com/google/pprof
4. 安装 graphviz
4. web 界面： pprof -http=:8080 cpu.prof

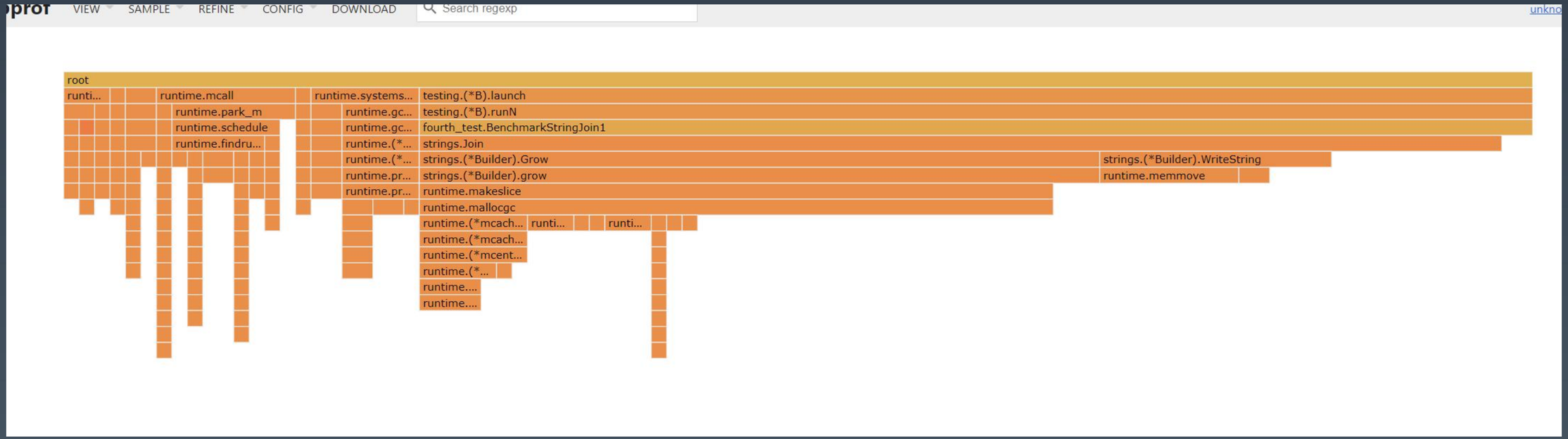


实践：性能测试 —— 调用链路


720ms
6ouath_test
BenchmarkStringJoin1
20ms (2.11%)
of 720ms (75.79%)
700ms
strings
Join
110ms (11.58%)
of 700ms (73.68%)
440ms
(inline)
150ms
(inline)
strings
(*Builder)
grow
30ms (3.16%)
of 440ms (46.32%)
strings
(*Builder)
WriteString
40ms (4.21%)
of 150ms (15.79%)
410ms
10ms
90ms
20ms
(inline)
runtime
memmove
90ms (9.47%)
strings
(*Builder)
copyCheck
20ms (2.11%)
410ms
10ms
10ms
80ms
runtime
mallocgc
230ms (24.21%)
of 410ms (43.16%)
runtime:
park_m
10ms (1.05%)
of 80ms (8.42%)
30ms
(inline)
10ms
10ms
(inline)
70ms
10ms
10ms
10ms
30ms
(inline)
10ms
10ms
10ms
runtime
nextFreeFast
30ms (3.16%)
runtime
publicationBarrier
10ms (1.05%)
runtime
releasem
10ms (1.05%)
runtime
acquirem
30ms (3.16%)
70ms
10ms
10ms
20ms
70ms
20ms
10ms
20ms
10ms
10ms
80ms
runtime
(*mcentral)
cacheSpan
10ms (1.05%)
of 70ms (7.37%)
50ms
10ms
10ms
30ms
10ms
10ms
30ms
runtime
(*mcentral)
grow
10ms (1.05%)
of 50ms (5.26%)
runtime
(*mspan)
refillAllocCache
10ms (1.05%)
runtime
checkRunqsNoP
10ms (1.05%)
40ms
10ms
10ms
70ms
10ms
30ms
30ms
10ms
30ms
40ms
10ms
10ms
10ms
10ms
runtime
stdcall6
30ms (3.16%)
40ms
10ms
10ms
60ms
10ms
20ms
30ms
runtime
memclrNoHeapPointers
40ms (4.21%)
10ms
(inline)
runtime
stdcall4
10ms (1.05%)
runtime
scanobject
10ms (1.05%)
runtime
(*consistentHeapStats)
release
10ms (1.05%)
10ms
60ms
10ms
20ms
30ms
(inline)
runtime
stop
10ms (1.05%)
60ms
10ms
20ms
30ms
60ms
10ms
20ms
30ms
runtime
addRangeg
removeGreaterEqual
10ms (1.05%)
runtime
stdcall3
20ms (2.11%)
30ms
(inline)
20ms
40ms

实践：性能测试 —— 火焰图


Wikipedia, the free...
Compose file versio...
astaxie/gotraining:...
Alibaba
pprof
VIEW
SAMPLE
REFINE
CONFIG
Top
Graph
Flame Graph
Peek
Source
Disassemble


pprof
VIEW
SAMPLE
REFINE
CONFIG
DOWNLOAD
Search regexp
root
runti...
runtime.mcall
runtime.park_m
runtime.schedule
runtime.findru...
runtime.systems...
runtime.gc...
runtime.gc...
runtime.(*...
runtime.(*...
runtime.pr...
runtime.pr...
testing.(*B).launch
testing.(*B).runN
fourth_test.BenchmarkStringJoin1
strings.Join
strings.(*Builder).Grow
strings.(*Builder).grow
runtime.makeslice
runtime.mallocgc
runtime.(*mcach...
runti...
runti...
runtime.(*mcach...
runtime.(*mcent...
runtime.(*...
runtime....
runtime....
strings.(*Builder).WriteString
runtime.memmove

实践：丐中丐 TDD——以 QueryBuilder 来举例

1. 站在用户的角度考虑，API 的输出和输出
2. 定义 API
3. 针对 API 写测试用例，此时只考虑正常情况
4. 写实现
5. 考虑各种乱七八糟的情况，写测试
6. 写实现，跳回第五步

链接

1. 代码：https://github.com/flycash/geekbang-go-camp