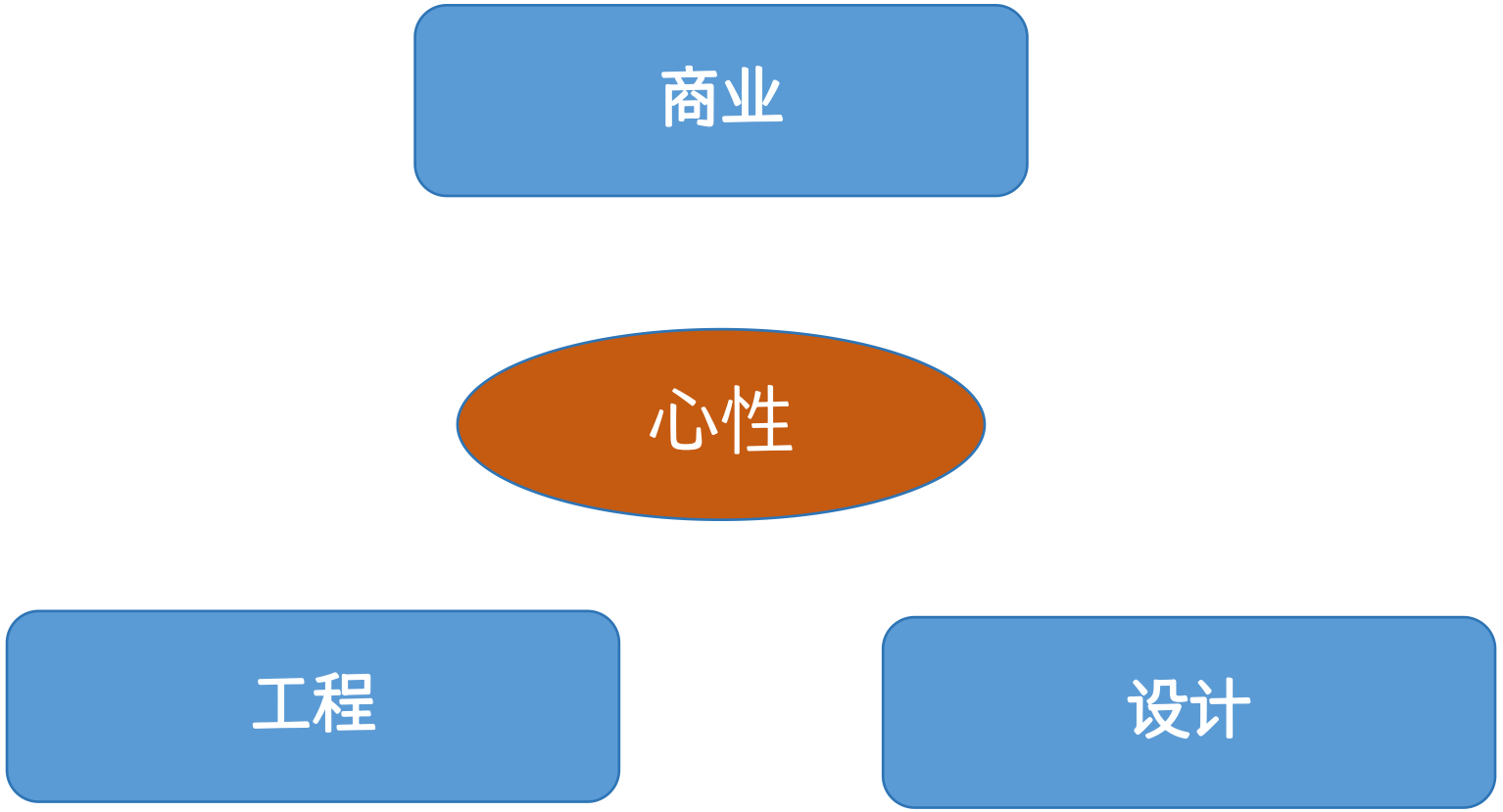
Go+、LLM+MCP与架构

@xushiwei

标题好凌乱，我想讲什么？

- 主题，其实是架构
- Go+、LLM+MCP 是实证的例子
- 这些例子看似无关，但其实是密切相关的
  - 并且，它们的架构逻辑有很大的共通之处

人才的能力象限



工程

- 工程是一门有关于如何“把事做成”的学问
- 工程可以和所有学科交叉（工程+X）

- 模块：将复杂问题拆解成独立子问题
- 版本：迭代逼近目标（节奏）
- 分工：团队协同

AI 时代下的个体能力象限

- 这个图不因 AI 时代到来而发生变化
- AI 时代下，个体能力变得前所未有的重要
  - **超级个体**，每个人都有机会做很大的事情
- 个体**能力的基础底座**不是智力，是**工程能力**

设计

- 设计不是指“产品包装”，而是一门有关于“决策”的学问
  - 要做什么事，以及把事做成什么样子

- 战略需要设计
- 组织需要设计
- 产品需要设计
- 架构需要设计

产品与架构

- 初级产品经理和初级架构师都喜欢**做加法**

- 但是**顶级产品经理**做的是**减法**
- 而**顶级架构师**做的是**乘法**

产品设计：少做加法，做减法

- **少就是指数级的多**
- 卖点越多的产品其实就是没有卖点
- 架构的**开闭原则**对产品同样适用
  - 产品要满足的需求可以是无限的，但**产品功能必须是封闭（有边界）**的
    - 例：Computer、LLM+MCP、Go+

Computer 如何实现无限可能?

- CPU
  - 有限的指令集：计算能力（大脑）
  - IO 指令：通过**数据交换**实现无限的扩展能力（及连接物理世界）
- 应用方
  - 指令序列（软件）：实现**领域适配**能力

LLM+MCP 如何实现无限可能？

- LLM+MCP = **AIPU**
  - LLM：算力 - 智力（绍兴师爷，君子动口不动手）
  - MCP：数据交换 - IO 能力（长工，具体干活的）

- 大模型应用
  - 实现**领域适配**能力

Go+ 如何实现无限可能？

Our vision is to enable everyone to become a builder of the world

- Go+ 的无限可能，不是理论上的无限可能，是**工程上的无限可能**
  - 理论上的无限可能：只要图灵完备，理论上做什么都可以
  - 工程上的无限可能：不要只是可能，要低成本的可能
    - 代价极高的可能性，从工程意义上来说就是不能

- 我们的目标是让 Go+ 成为**工程意义上无所不能**的语言
  - **做什么都很方便**（低门槛）
  - **做什么的代价都极低**（指要做到**所花的钱**）

Go+ 如何实现无限可能？

- **不是加功能，是减功能**
  - Go Spec 是今天工程上最精简的语法集（座右铭：少就是指数级的多）
  - 但 Go Spec 还可以进一步精简，于是有了 **Go+ Mini Spec**
    - Go+ Mini Spec 是 Go+ 推荐的工程最佳实践

- **不是自立门户，是融合的超级生态**
  - Go+ 当前已经支持 import Go、C/C++、Python 的包，就好像它们天生就用同一门语言写的一样（未来还将支持 import JavaScript 的包）
  - **Go、C/C++、Python、JavaScript 的资产总值约等于软件工程 55 年历史的所有资产的总值**（增加其他语言的支持几乎不会再提升资产总额）
    - Why?

怎么可能?

- 这世上会有这么 NB 的语言么?
- **Go+ 如何做到将软件工程的几乎所有资产融合在一起的?**

什么是架构设计?

- 架构从手法上就是模块拆解
- 连接与组合
- 实体 (Entity) 的边界问题

架构设计：少做加法，做乘法

- 模块的**规格高于实现**
  - 时序图是基于规格串起来的用户故事，用伪代码比UML图成本更低

- 假设你要做一个网站，把页面1交给A，页面2交给B，然后开干，这是典型的一次做加法的过程

- 把整个网站需求分解为 A、B、C、D、E 等完全正交无耦合、甚至和你的网站原始需求不直接挂钩的**通用模块**，用极短的桥接代码1把这些通用模块组装起来完成页面1，桥接代码2完成页面2，这才是做乘法
  - 而判断乘法做的好不好的标准非常简单：**桥接代码的代码量越少**，乘法的威力越大，你的**架构设计的能力也就越强**

何为乘法?

- **每个模块都是独立业务**

- 它独立看有更大的需求边界
- 它独立看很有前景（很多人都有类似的需要）

- **模块价值 = 需求规模 * 模块实现成本**

Go+ := Go+语言 (gop) * Go编译器 (llgo)

- Go+ 语言 (gop)：**专注于表达**
  - 更简洁、易于掌握的语法
  - 更少的推荐语法集（Go+ Mini Spec）
  - 语言本身已达到工业级，未来重点完善工具链（有但品质还有待提高）

- Go 编译器 (llgo)：**专注于能力边界的扩张**
  - Go 侧重点在服务端，但 Go+ 并不满足于此，它要的是无限可能
  - 支持 import Go 的包
  - 支持 import C/C++ 的包（已经支持，且已经基本达到工业级）
  - 支持 import Python 的包（已经支持，在 Go+ v1.5 版本将达到工业级）
  - 支持 import JavaScript 的包（预计要等到 Go+ v1.7 之后）

Go+: import C library

c.go

```
//go:linkname Printf C.printf
func Printf(format *Char, __llgo_va_list ...any) Int
```

hello.gop

demo > _llgo > hellollgo > hello.gop

```
1  import "c"
2
3  c.printf c"Hello, llgo!\n"
4  c.fprintf c.Stderr, c"Hi, %6.1f\n", 3.14
5
```

qsort.gop

demo > _llgo > qsort > qsort.gop

```
1  import (
2      "c"
3      "unsafe"
4  )
5
6  a := [100, 8, 23, 2, 7]
7  c.qsort unsafe.Pointer(&a[0]), 5, unsafe.Sizeof(0), (a, b) => {
8      return c.Int(*(*int)(a) - *(*int)(b))
9  }
10 for v <- a {
11     c.printf c"%d\n", v
12 }
```

C/C++ 的能力
Go+ 的语法
没有 cgo

sqlitedemo.gop

demo > _llgo > sqlitedemo > sqlitedemo.gop

```
import (
    "c"
    "c/sqlite"
    "c/os"
)

func check(err sqlite.Errno, db *sqlite.Sqlite3, at string) {
    if err != sqlite.OK {
        c.printf c"==> %s Error: (%d) %s\n", c.allocaCStr(at), err, db.errmsg
        c.exit 1
    }
}

func checkDone(err sqlite.Errno, db *sqlite.Sqlite3, at string) {
    if err != sqlite.Done {
        check err, db, at
    }
}

os.remove c"test.db"

db, err := sqlite.open(c"test.db")
check err, db, "sqlite: Open"

err = db.exec(c"CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)", nil, nil, nil)
check err, db, "sqlite: Exec CREATE TABLE"

stmt, err := db.prepareV3("INSERT INTO users (id, name) VALUES (?, ?)", 0, nil)
```

cpphello.gop

demo > _llgo > cpphello > cpphello.gop

```
import "cpp/std"

println std.str("Hello world").str
```

互操作也很方便

可以不用 Go 标准库
只用 C 标准库
虽然并不推荐

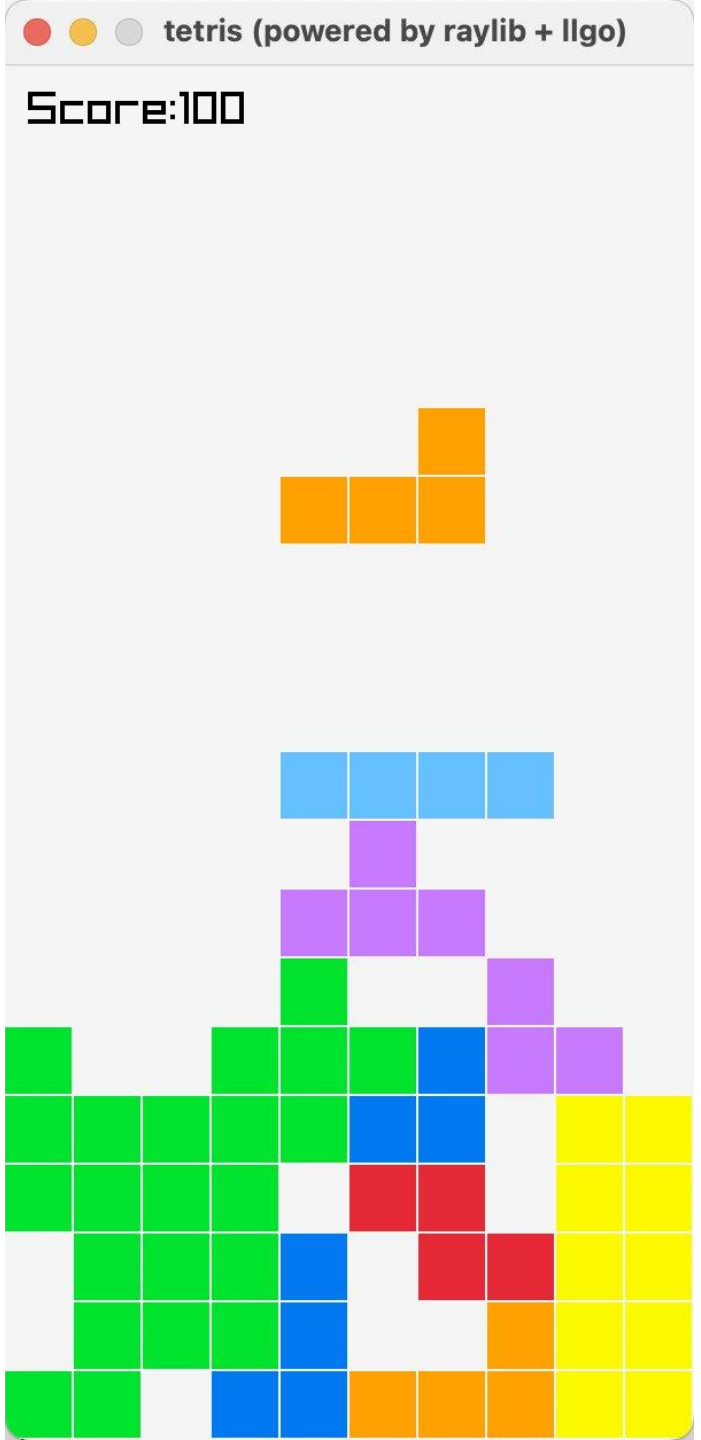
Game Powered by raylib

tetris.gop

testdata > _llgo > tetris > tetris.gop > ...

run main package

```
import (
    "c"
    "c/raylib"
)

const (
    BOARD_WIDTH  = 10
    BOARD_HEIGHT = 20
    BLOCK_SIZE   = 30

    SCREENWIDTH  = 300
    SCREENHEIGHT = 600
)

const MAX_BLOCKS = 4

type Shape struct {
    Blocks [MAX_BLOCKS]raylib.Vector2
    Color  raylib.Color
}

var SHAPES = []Shape{
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 0, Y: 0}, {X: 1, Y: 0}, {X: 2, Y: 0},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 0, Y: 0}, {X: 1, Y: 0}, {X: 0, Y: 1},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 1, Y: 0}, {X: 0, Y: 1}, {X: 1, Y: 1},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 1, Y: 0}, {X: 2, Y: 0}, {X: 0, Y: 1},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 0, Y: 0}, {X: 1, Y: 0}, {X: 1, Y: 1},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 0, Y: 0}, {X: 0, Y: 1}, {X: 1, Y: 1},
    {Blocks: [MAX_BLOCKS]raylib.Vector2{{X: 2, Y: 0}, {X: 0, Y: 1}, {X: 1, Y: 1},
}

var board [BOARD_HEIGHT][BOARD_WIDTH]raylib.Color
```



Go+ 对 C/C++ 的支持已经定型

- 对 C 字符串的内建支持：c"Hello"
  - c.str 是一条 LLGo 的编译器指令
  - 不是普通函数，不是将 Go 字符串转成 C 字符串

c.go

c > c.go > ...

```
78    //go:linkname Str llgo.cstr
79    func Str(string) *Char
```

- 没有内建支持 C++ 字符串，没有 cpp"Hello"
  - std::string 在 C++ 世界并不是共识，有很多自定义的 string
  - 目前 cpp.str 是一个普通函数，是将 Go 字符串转为 C++ 字符串

string.go

cpp > std > string.go > ...

```
112   // Str creates a constant C++ std::string object.
113   func Str(v string) *String {
```

Go+: import Python library

print.gop

testdata > _llgo > pyprint > print.gop > ...

run main package

```
1  import (
2      "py"
3      "py/std"
4  )
5
6  x := py.float(3.14)
7  std.print(x)
```

OUTPUT DEBUG CONSOLE TERMINAL TEST RESUL

```
(llgo_env) xushiwei@xushiweis-Laptop pyprint %
3.14
```

tensor.gop

testdata > _llgo > pytensor > tensor.gop > ...

run main package

```
1  import (
2      "py"
3      "py/std"
4      "py/torch"
5  )
6
7  data := py.list(
8      py.list(1.0, 2.0),
9      py.list(3.0, 4.0),
10 )
11 x := torch.tensor(data)
12 std.print(x)
```

OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS

```
(llgo_env) xushiwei@xushiweis-Laptop pytensor % gop run .
tensor([[1., 2.],
        [3., 4.]])
```

Go+ 对 Python 的支持还没有定型

- **C/C++、Python 在 Go+ 中和 Go 一样是一等公民**
  - 只要是足够大的共识，它就应该被支持
- 已经内建支持 Python 字符串：py"Hello"
  - 它不是 Go 字符串转 Python 字符串，而是 Python 字符串字面量
- 以下内容将在接下来的 **Go+ v1.5** 中明确下来
  - 更普遍的 Python 字面量？
    - 比如让 std.print 3.14 等价于 std.print py.float(3.14)
  - 更好用的 Python List？
    - 不要用 py.list(v1, v2, ...)，而是用 [v1, v2, ...]
      - 怎么区分 Python List 和 Go List

从 Go+ 的实现看架构拆解

- Go+ 产品定义中的乘法
  - Go+ := gop * llgo，这是 **Go+ 中最大的乘法**
    - **gop 专注表达，llgo 专注能力边界的扩张**
  - llgo := Go * C * Python
    - 我以前这里都用 + 表达，但从架构视角来说用 * 含义更准确（为啥？）
      - **因为 Go、C/C++、Python 在 Go+ 中并不是孤立的，而是有机的整体**
  - Go+ 编译器会生成 .go 文件，并鼓励入库
    - 不必担心 Go+ 如果有一天不行了你的资产无法保障
    - 可以**复用 Go 的工具链**（比如文档站、Debugger、Profile 等）
- gop 和 llgo 的实现，也大量使用乘法

- Go+ 源代码 => Go+ AST
  - gop/scanner + gop/parser + gop/ast（为什么是 3 个？）
  - gop/tpl/scanner + gop/tpl + gop/ast（另一种可能的实现路径）
  - gop/tpl 是与 **Go+** 语言无关的独立业务

模块分资产型和过程型
资产型往往是过程型的环境(输入/输出)
资产型通常更重要(规格变化的影响面)

- Go+ AST => Go AST => Go 源代码 => 可执行程序
  - gop/cl + gogen + go compiler (go, llgo or tinygo)
  - 到 Go 源代码不是必须的，但是是当前最合适的（为什么这么说？）
  - gogen 是独立于 **Go+** 语言的业务
  - gop/cl 和 **Go+** 语言密切相关，但代码量尽可能少

gop/cl 和 gogen 的边界是什么?

gop / cl / **expr.go**

Code | Blame | 1672 lines (1554 loc) · 40.7 KB

```
413     func compileUnaryExpr(ctx *blockCtx, v *ast.UnaryExpr, twoValue bool) {
414             compileExpr(ctx, v.X)
415             ctx.cb.UnaryOp(gotoken.Token(v.Op), twoValue, v)
416     }
417
418     func compileBinaryExpr(ctx *blockCtx, v *ast.BinaryExpr) {
419             compileExpr(ctx, v.X)
420             compileExpr(ctx, v.Y)
421             ctx.cb.BinaryOp(gotoken.Token(v.Op), v)
422     }
```

- Go 源代码 => Go AST => Go SSA
  - go/scanner+parser+ast + tools/go/packages + tools/go/ssa
  - 这些模块都不是我们写的

- Go SSA => LLVM SSA => 可执行程序
  - llgo/cl + llgo/ssa + goplus/llvm + clang（只有非常局部的一块是我们的）
  - llgo/ssa 是独立于 **LLGo** 编译器的业务
  - llgo/cl 和 **LLGo** 编译器密切相关，但代码量尽可能少

llgo/cl 和 llgo/ssa 的边界是什么？

llgo / cl / **compile.go**

Code | Blame | 1149 lines (1070 loc) · 28.2 KB

```
601  func (p *context) compileInstrOrValue(b llssa.Builder, iv instrOrValue, asValue bool) (ret llssa.Expr) {
611          case *ssa.BinOp:
612                  x := p.compileValue(b, v.X)
613                  y := p.compileValue(b, v.Y)
614                  ret = b.BinOp(v.Op, x, y)
615          case *ssa.UnOp:
616                  x := p.compileValue(b, v.X)
617                  if v.Op == token.ARROW {
618                          ret = b.Recv(x, v.CommaOk)
619                  } else {
620                          ret = b.UnOp(v.Op, x)
621                  }
```

从 Go+ 的实现看架构拆解

- 一个逆直觉的重要结论：

- **模块切分，通常和需求并不形成对应关系**
  - 如果对应，往往反而说明模块划分和团队分工是有问题的

- 需求分析要做什么？
  - 需求未来可能的发展方向预判（防止过度设计）
  - **洞察需求的内在逻辑关联（模块切分的基础）**

Go+ v1.5 正在做哪些事情?

- 大模型的支持：**AIPU** := LLM + MCP
- Python 的内建支持：让 Python 进一步成为 Go+ 的一等公民

Go+ MCP

- **https://github.com/goplus/mcp**
  - Go+ MCP v0.8.2 已发布

- 当前提供了 MCP Server、MCP Server Test 框架

main_mcp.gox

demo > hello > main_mcp.gox

```
1  server "Tool Demo 🚀", "1.0.0"
2
```

hello_tool.gox

demo > hello > hello_tool.gox

```
1  tool "helloWorld", => {
2      description "Say hello to someone"
3      string "name", => {
4          required
5          description "Name of the person to greet"
6      }
7  }
8
9  name, ok := ${name}.(string)
10 if !ok {
11     panic "name must be a string"
12 }
13
14 return text("Hello, ${name}!")
```

hello_mtest.gox

demo > hello > hello_mtest.gox

```
1  mock new(MCPApp)
2
3  initialize nil
4  ret {}
5
6  list "tools"
7  ret {…
26 }
27
28 call "helloWorld", {"name": "Ken"}
29 ret {
30     "content": [{"type": "text", "text": "Hello, Ken!"}],
31 }
```

可以 mock 一个 MCP Server
并不需要真监听一个端口

code_review_prompt.gox

demo > prompt > code_review_prompt.gox

```
prompt "codeReview", => {
    description "Code review assistance"
    arg "prNumber", => {
        required
        description "Pull request number to review"
    }
}

prNumber := ${prNumber}
if prNumber == "" {
    panic "prNumber is required"
}

return "Code review assistance", [
    prompt(RoleUser, text("You are a helpful code reviewer. Review the changes and provide c
    prompt(RoleAssistant, embedded({
        URI:      "git://pulls/${prNumber}/diff",
        MIMEType: "text/x-diff",
        Text:     "diff",
    })),
]
```

code_review_mtest.gox

demo > prompt > code_review_mtest.gox

```
testServer new(MCPApp)

initialize nil
ret {}

prompt "codeReview", {"prNumber": "123"}
ret {
    "description": "Code review assistance",
    "messages": [
        {
            "content": {
                "text": "You are a helpful code reviewer. Review the changes and provide con
                "type": "text",
            },
            "role": "user",
        },
        {
            "content": {
                "resource": {
                    "mimeType": "text/x-diff",
                    "text":     "diff",
                    "uri":      "git://pulls/123/diff",
                },
                "type": "resource",
            },
            "role": "assistant",
        },
    ],
}
```

真起来一个 MCP Server
会使用一个随机端口

关于 Go+ MCP，你可能的疑问

- Go+ 源代码文件名不是 .gop 么，怎么来了个 .gox 呢？
- server、tool、prompt 这些是啥，Go+ 语言的内置指令么？

Go+ classfile

- .gox 是一种特殊的 Go+ 源代码叫 classfile，它定义了一个类
  - 假设有一个 Rect.gox，代码如图：

```
var (
    Width, Height int
)

func Area() int {
    return Width * Height
}
```

    - 它定义了 Rect 类，有
      - Width、Height 两个成员变量
      - 有 Area 这个成员函数
    - 它等价于以下 Go 代码
      - Go+ 并不鼓励这样写代码，**所有需要定义成员函数的地方，都应该用 classfile**

```
type Rect struct {
    Width, Height int
}

func (this *Rect) Area() int {
    return this.Width * this.Height
}
```

classfile 的全局代码不是在 main 函数中
而是在 (*Rect).Main 函数中，也是成员函数

```
rect := &Rect{10, 20}
echo rect.area
```

Go+ classfile

- 有两类 classfile
  - 普通 classfile，例如前面的 Rect.gox
    - 这种很容易理解
  - 领域 classfile，例如前面的：
    - main_mcp.gox
    - hello_tool.gox
    - code_review_prompt.gox
    - hello_mtest.gox
    - code_review_mtest.gox

什么是领域 classfile?

- 领域 classfile 由一个 project class + 任意多个 work class 构成
  - 每个 project class 和 work class 都是一个 classfile 文件
  - 但它们彼此相关，构成一个有机整体

- 领域 classfile 之所以叫领域 classfile 是因为它有**基类**
  - **基类定义了领域知识**
    - server、tool、prompt 这些都是由基类定义的，所以在 classfile 中可以用它们

- work class 除了基类，也自动包含了一个 project class 的指针
  - **它除了可以用基类定义的领域知识，也可以用本领域的全局知识**
    - main_mcp.gox / hello_tool.gox / code_review_prompt.gox 构成一个整体，即 MCP classfile
    - hello_mtest.gox / code_review_mtest.gox 构成一个整体，即 MCP Test classfile
      - 注意 project class 用户没有实现，此时用默认的 project class 代码

架构师的武器库是什么？

- 架构思维？
- 设计模式？
- ...

武器库的定义

- 它不能是抽象的，必须是一个实体（entity）
  - 架构思维在人的脑子里，它不可复制
  - 设计模式也在人的脑子里，它不可复制

- 我们必须找到一种设施，它真切存在，可被传承

classfile 背后的架构思想

- 这个世界的领域，其实并不多
  - AI Programming
  - CLI Programming
  - GUI Programming
  - Game Programming
  - Web/Server Programming
  - Text Processing Programming
  - ...

- STEM Education: spx: A Go+ 2D Game Engine
- AI Programming: mcp: A Go+ implementation of the Model Context Protocol (MCP)
- AI Programming: mcptest: A Go+ MCP Test Framework
- Web Programming: yap: Yet Another HTTP Web Framework
- Web Programming: yaptest: A Go+ HTTP Test Framework
- Web Programming: ydb: A Go+ Database Framework
- CLI Programming: cobra: A Commander for modern Go+ CLI interactions
- CLI Programming: gsh: An alternative to writing shell scripts
- Unit Test: test: Unit Test

- 每个领域的方法论，就算刚开始没有共识，几年或十几年终归会有共识
  - 如何沉淀这些共识？
  - classfile 是其中最关键的基础设施

Q & A