

GO

Go1ang微服务 开发

许志宏





目录

01

环境篇

02

框架篇

03

常用包篇

04

中间件篇

05

应用篇

06

经验篇



微服务现象





01 环境篇



01 | 依赖版本管理

`go module`的直接原型是`vgo`，`vgo`是官方对包版本管理的一个新的实验，它是一个可替换`go`命令的工具，并增加了包版本管理功能；GO v1.12版本正式将`vgo`集成到`go`命令中，GO v1.13版本开始成为默认的版本管理工具。



解决什么问题?

1、版本依赖管理

如：模块foo依赖模块baz@v1.0.1，模块bar依赖模块baz@v1.0.2，项目foobar依赖foo和bar

2、脱离对GOPATH的依赖

项目源代码无须放在\$GOPATH/src目录下

3、无须使用vender目录

第三方包统一下载到\$GOPATH/pkg/mod目录下



常用命令 初始化:

// 进入module目录，执行如下命令，生成go.mod文件

```
go mod init <modulename>
```

整理包依赖:

// 整理项目包依赖，删除未使用的依赖

```
go mod tidy -v
```

依赖包导入链:

// 可查看直接或间接包的导入链

```
go mod why -m <pkg>
```



查看构建项目使用的依赖包版本号:

```
go list -m all | grep <pkg>
```

包依赖关系:

// 包依赖谁, 谁依赖此包

```
go mod tidy -v
```




更新包命令

查看包有那些版本:

// 如: `go list -m -versions github.com/gin-gonic/gin`

`go list -m -versions <pkg>`

更新到修订版本:

`go get -u=patch <pkg>`

更新到指定版本:

`go get <pkg>@vX.Y.Z`

更新到最新版本:

`go get -u <pkg>`



本地包依赖

包存放在私有仓库或者本地，代码托管网站上无法找到，可使用`replace`语句替换依赖。如：

// 在`go.mod`文件中添加`replace`语句

`go.mod`:

```
replace github.com/robteix/testmod => ./local/codes/src/testmod
```



Vender模式

go module提供了vendor模式，用于将项目依赖包纳入版本控制中。

第一步：创建vendor目录，将依赖下载至此目录

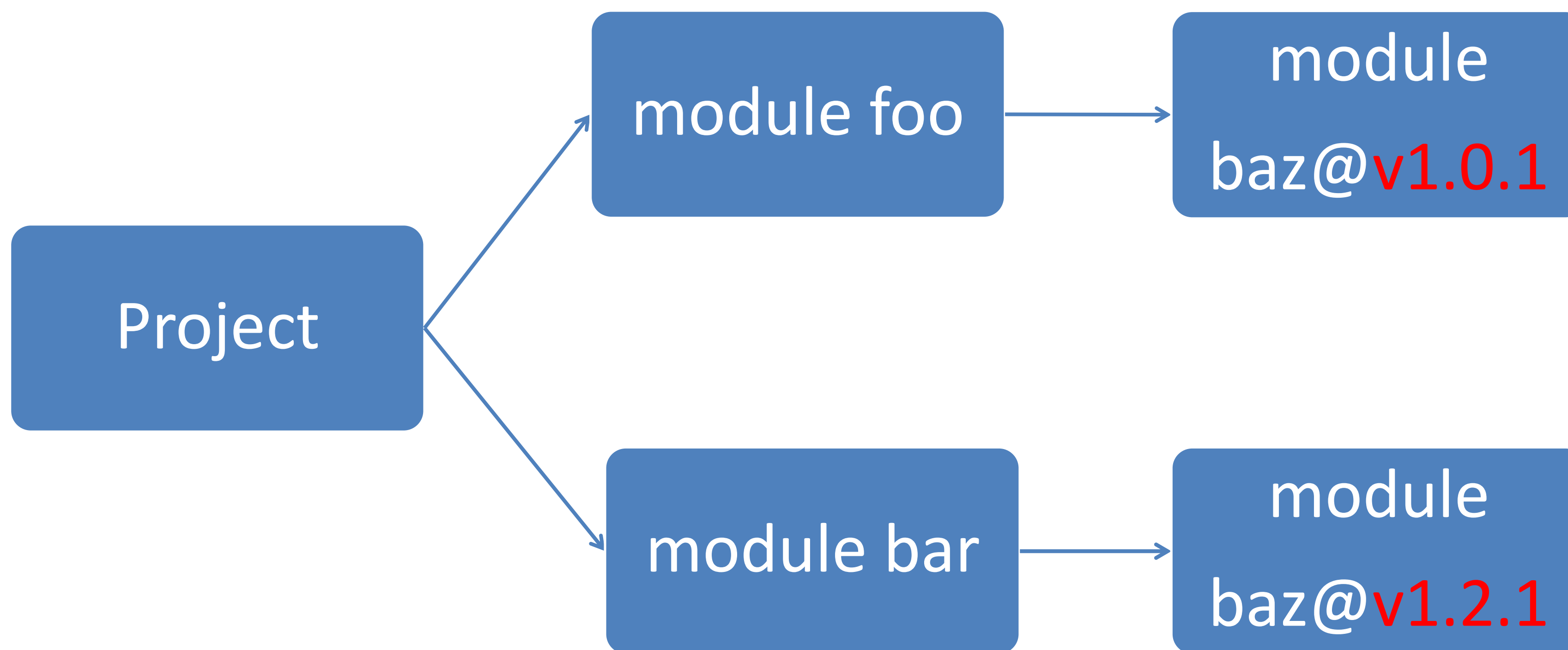
```
// go mod vendor
```

第二步：编译时从vendor目录查找依赖

```
// go build -mod vendor
```




最小版本选择





go.mod

go.mod用于定义go模块，文件中记录模块依赖的**最小集合**。除指定依赖列表外，还可以指定排除或替换的**直接依赖**。



go.mod: Indirect标识

1、手动go get的模块。

2、直接依赖未使用go module(缺少go.mod文件)。

如：foo直接依赖bar，bar直接依赖baz，baz会出现在foo的go.mod文件中



go.mod: 伪版本

go.mod文件中有时会出现格式如下的依赖项:

```
require github.com/fvbock/endless v0.0.0-20170109170031-447134032cb6
```

此现象表示包没有tag, go mod基于主分支最近一次提交的时间和commithash来生成伪版本。



go.sum

go.sum文件详细罗列了所有直接和间接依赖的模块，它协助go.mod文件完成编译，用于完成包的效验。

文件格式：

<module> <version> <hash>

<module> <version>/go.mod <hash>

如：

github.com/BurntSushi/toml v0.3.1

h1:WXkYYl6Yr3qBf1K79EBnL4mak0OimBfB0XUf9VI28OQ=

github.com/BurntSushi/toml v0.3.1/go.mod

h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5slBksp03qcyfWMU=



统一代码风格

本地修改代码上传到版本控制系统前，应将代码格式化为官方统一的风格，可通过两个命令完成此功能：`gofmt`、`goimports`。

`goimports`除完成`gofmt`命令功能外，还能统一`import`的顺序。



goimports

安装与使用

安装:

- 1、`go get golang.org/x/tools/cmd/goimports`
- 2、`cd $GOPATH/pkg/golang.org/x/tools/cmd/goimports`
- 3、`go install`

使用:

- 1、格式化单个文件

```
goimports -w filename
```

- 2、格式化目录下的所有文件

```
goimports -w dirname
```



bin文件加入编译信息

编译bin文件时可加入编译时间、golang版本号、操作系统类型、版本号等编译信息。

如，`docker version`命令输出的相关信息：

```
[root@localhost ~]# docker version
```

```
Version: 19.03.13
```

```
API version: 1.40
```

```
Go version: go1.13.15
```

```
Git commit: 4484c46d9d
```

```
Built: Wed Sep 16 17:03:45 2020
```

```
OS/Arch: linux/amd64
```



01 | 编译

main.go:

```
var (BUILTTIME, GOVERSION, OSARCH string)
```

```
func main() {
```

```
    fmt.Println(BUILTTIME, GOVERSION, OSARCH)
```

```
}
```

// 编译

```
go install -ldflags "-X 'main.BUILTTIME=`date`' -X main.GOVERSION=`go version | awk
```

```
'{print $3}`' -X 'main.OSARCH=`uname -s -m`'" main.go
```

// 输出

```
2020年 12月 17日 星期四 14:01:39 CST go1.13.6 Linux x86_64
```




静态代码分析: **golangci-lint**

默认启用以下linter:

deadcode: 发现未使用的代码

errcheck: 返回的error未做检查

gosimple: 代码是否可以简化

govet: 可疑的代码结构, 如: printf调用参数与格式字符串个数不一致

structcheck: 未使用的结构体字段

unused: 未使用的常量、变量、函数及类型

varcheck: 未使用的全局变量和常量



项目编译脚本

```
#!/bin/env bash

export GOROOT="/usr/local/go1.13.6/"
export GOPATH="/codes/golang/"
echo "GOROOT="$GOROOT
echo "GOPATH="$GOPATH

if [ "$1" == "test" ]; then
    go test ./...
    exit
fi

date=`date "+%Y-%m-%d %H:%I:%S"`
goverversion=`go version | awk '{print $3}'`
osarch=`uname -s -m`
ldflags="-X main._BUILTTIME_=$date -X main._GOVERSION_=$goverversion -X main._GOVERSION_=$osarch"
echo "ldflags="$ldflags

echo -e "\nFormatting code..."
goimports -w ./

echo -e "\nGolangci-lint code..."
golangci-lint run

echo "Install..."
#$GOROOT/bin/go install -race -ldflags "$ldflags" ./

echo -e "\nEnd."
```



02 框架篇



框架优势

性能高

采用压缩前缀树，路由查找速度快
代码精简，未使用反射

扩展性好

中间件与业务代码隔离
简单易用、上手快

功能丰富

分组路由管理
丰富的render方法
参数绑定及支持自定义



实例

```
package main

import "github.com/gin-gonic/gin"

func main() {
    // 创建Engine ,绑定Logger、Recovery中间件
    r := gin.Default()

    // 添加路由, 以及路由处理函数
    r.GET(relativePath: "/ping", func(c *gin.Context) {
        c.JSON(code: 200, gin.H{
            "message": "pong",
        })
    })

    r.Run()
}
```




路由接口

HandlerFunc是路由处理函数类型定义。可通过实现IRoutes接口来自定义路由引擎。

```
type HandlerFunc func(*Context)

// IRoutes defines all router handle interface.
type IRoutes interface {
    Use(...HandlerFunc) IRoutes

    Handle(string, string, ...HandlerFunc) IRoutes
    Any(string, ...HandlerFunc) IRoutes
    GET(string, ...HandlerFunc) IRoutes
    POST(string, ...HandlerFunc) IRoutes
    DELETE(string, ...HandlerFunc) IRoutes
    PATCH(string, ...HandlerFunc) IRoutes
    PUT(string, ...HandlerFunc) IRoutes
    OPTIONS(string, ...HandlerFunc) IRoutes
    HEAD(string, ...HandlerFunc) IRoutes
}
```



路由注册

路由默认区分大小写

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.New()

    // 注册http Get方法路由
    r.GET(relativePath: "/uri1", func(c *gin.Context) {})

    // 注册http Post方法路由
    r.POST(relativePath: "/uri2", func(c *gin.Context) {})

    // 注册路由并绑定到所有http方法
    r.Any(relativePath: "/uri3", func(c *gin.Context) {})

    r.Run()
}
```



02 | gin框架-动态路由

冒号(:)属于精准匹配,
只能匹配一个参数,
如/user/:name匹配以下uri:

/user/zhangsan

不匹配以下uri:

/user/zhang/san

/user/

/user

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    r := gin.Default()

    r.GET(relativePath: "/user/:name", func(c *gin.Context) {
        name := c.Param(key: "name")
        c.String(http.StatusOK, format: "Hello %s", name)
    })

    r.Run()
}
```



02 | gin框架-动态路由

星号(*)表示模糊匹配,

如/user/*name匹配以

下uri:

/user/zhangsan

/user/zhang/san

/user/

/user

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    r := gin.Default()

    r.GET("/user/*name", func(c *gin.Context) {
        name := c.Param("name")
        c.String(http.StatusOK, "Hello %s", name)
    })

    r.Run()
}
```



02 | gin框架-路由分组

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()

    v1 := r.Group( relativePath: "/v1" ) // /v1
    {
        v1.POST( relativePath: "/login" ) // /v1/login
    }

    v2 := r.Group( relativePath: "/v2" ) // /v2
    {
        v2.POST( relativePath: "/login" ) // /v2/login
    }

    r.Run()
}
```




路由分组

嵌套

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()

    // /v1
    v1 := r.Group( relativePath: "v1" )
    {
        v1User := v1.Group( relativePath: "/user" )
        // /v1/user/login
        v1User.GET( relativePath: "/login", func(c *gin.Context) {} )
    }

    r.Run()
}
```



02 | gin框架-前缀路由

前缀路由是指能匹配以某字段开头的所有uri，如路由“/user ”能匹配 /user/login、 /user/logout、 /user/home/index等。
我们可以通过gin路由+反射来实现。



02 | gin框架-前缀路由

步骤一：添加Any路由和服务Controller类型

```
type ServiceController struct {  
    w http.ResponseWriter  
    req *http.Request  
    ctx *gin.Context  
}  
  
func main() {  
    r := gin.Default()  
    r.Any(relativePath: "/user/*action", ServiceHandler)  
    r.Run()  
}
```



02 | gin框架-前缀路由

步骤二：实现gin路由处

理函数ServiceHandler

```
}func ServiceHandler(c *gin.Context) {  
    w := c.Writer  
    req := c.Request  
  
    pathInfo := strings.Trim(req.URL.Path, cutset: "/")  
    paths := strings.Split(pathInfo, sep: "/")  
    actions := make([]string, len(paths))  
    for _, v := range paths {  
        actions = append(actions, strings.Title(v))  
    }  
    action := strings.Join(actions, sep: "")  
  
    service := &ServiceController{w: w, req: req, ctx: c}  
    controller := reflect.ValueOf(service)  
  
    method := controller.MethodByName(action + "Handler")  
    if method.IsValid() {  
        method.Call([]reflect.Value{})  
    } else {  
        c.AbortWithStatus(http.StatusNotFound)  
    }  
}
```



02 | gin框架-前缀路由

步骤三：实现单个路由处理函数，如/user/login路由的处理函数是UserLoginHandler

```
func (srv *ServiceController) UserLoginHandler() {  
    srv.ctx.String( code: 200, format: "success")  
}
```




02 | gin框架-路由注册原理

gin使用压缩前缀树来保存路由，每种httpMethod单独使用一棵树，即：Any路由注册方法会生成9棵前。假设有以下路由：

GET /user

GET /user/abc

GET /user/abcd

GET /user/1234

对应的前缀树生成步骤如下：



02 | gin框架-路由注册原理

步骤一：添加/user路由, 此时树只有一个节点

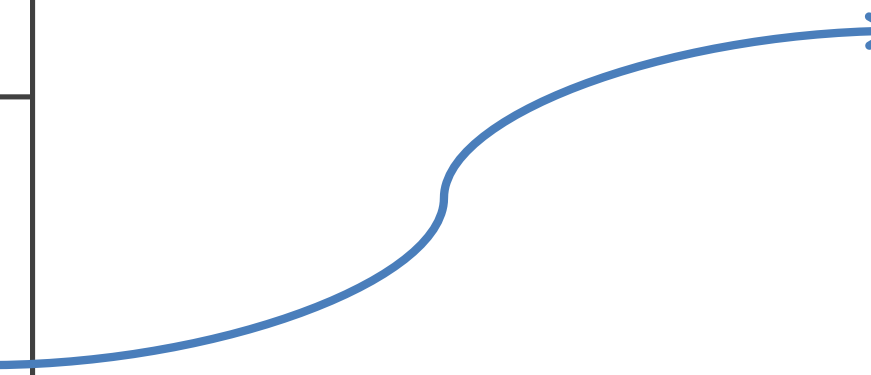
	node
	<pre>path: "/user", indices: "", children: nil, priority: 1, nType: root, maxParams: 0, wildChild: false, fullPath: "/user",</pre>



02 | gin框架-路由注册原理

步骤二：添加/user/abc路由, 树变成两个节点

	node
	<pre>path: "/user", indices: "/", children: []*node{}, priority: 2, nType: root, maxParams: 0, wildChild: false, fullPath: "/user",</pre>



	node
	<pre>path: "/abc", indices: "", children: nil, priority: 1, nType: static, maxParams: 0, wildChild: false, fullPath: "/user/abc",</pre>



02 | gin框架-路由注册原理

步骤三：添加/user/abcd路由

	node
	<pre>path: "/user", indices: "/", children: []*node{}, priority: 3, nType: root, maxParams: 0, wildChild: false, fullPath: "/user",</pre>

	node
	<pre>path: "/abc", indices: "d", children: []*node{}, priority: 2, nType: static, maxParams: 0, wildChild: false, fullPath: "/user/abc",</pre>

	node
	<pre>path: "d", indices: "", children: nil, priority: 1, nType: static, maxParams: 0, wildChild: false, fullPath: "/user/abcd",</pre>



02 | gin框架-路由注册原理

步骤四：添加/user/1234路由

node
path: <code>"/user"</code> , indices: <code>"/"</code> , children: <code>[]*node{}</code> , priority: <code>4</code> , nType: <code>root</code> , maxParams: <code>0</code> , wildChild: <code>false</code> , fullPath: <code>"/user"</code> ,

node
path: <code>"/"</code> , indices: <code>"a1"</code> , children: <code>[]*node{}</code> , priority: <code>3</code> , nType: <code>static</code> , maxParams: <code>0</code> , wildChild: <code>false</code> , fullPath: <code>"/user/"</code> , handlers: <code>nil</code> ,

node
path: <code>"abc"</code> , indices: <code>"d"</code> , children: <code>[]*node{}</code> , priority: <code>2</code> , nType: <code>static</code> , maxParams: <code>0</code> , wildChild: <code>false</code> , fullPath: <code>"/user/abc"</code> ,

node
path: <code>"d"</code> , indices: <code>""</code> , children: <code>nil</code> , priority: <code>1</code> , nType: <code>static</code> , maxParams: <code>0</code> , wildChild: <code>false</code> , fullPath: <code>"/user/abcd"</code> ,

node
path: <code>"1234"</code> , indices: <code>""</code> , children: <code>[]*node{}</code> , priority: <code>1</code> , nType: <code>static</code> , maxParams: <code>0</code> , wildChild: <code>false</code> , fullPath: <code>"/user/1234"</code> ,



动态路由比静态路由稍复杂，以下动态路由对应的前缀树在下页PPT。

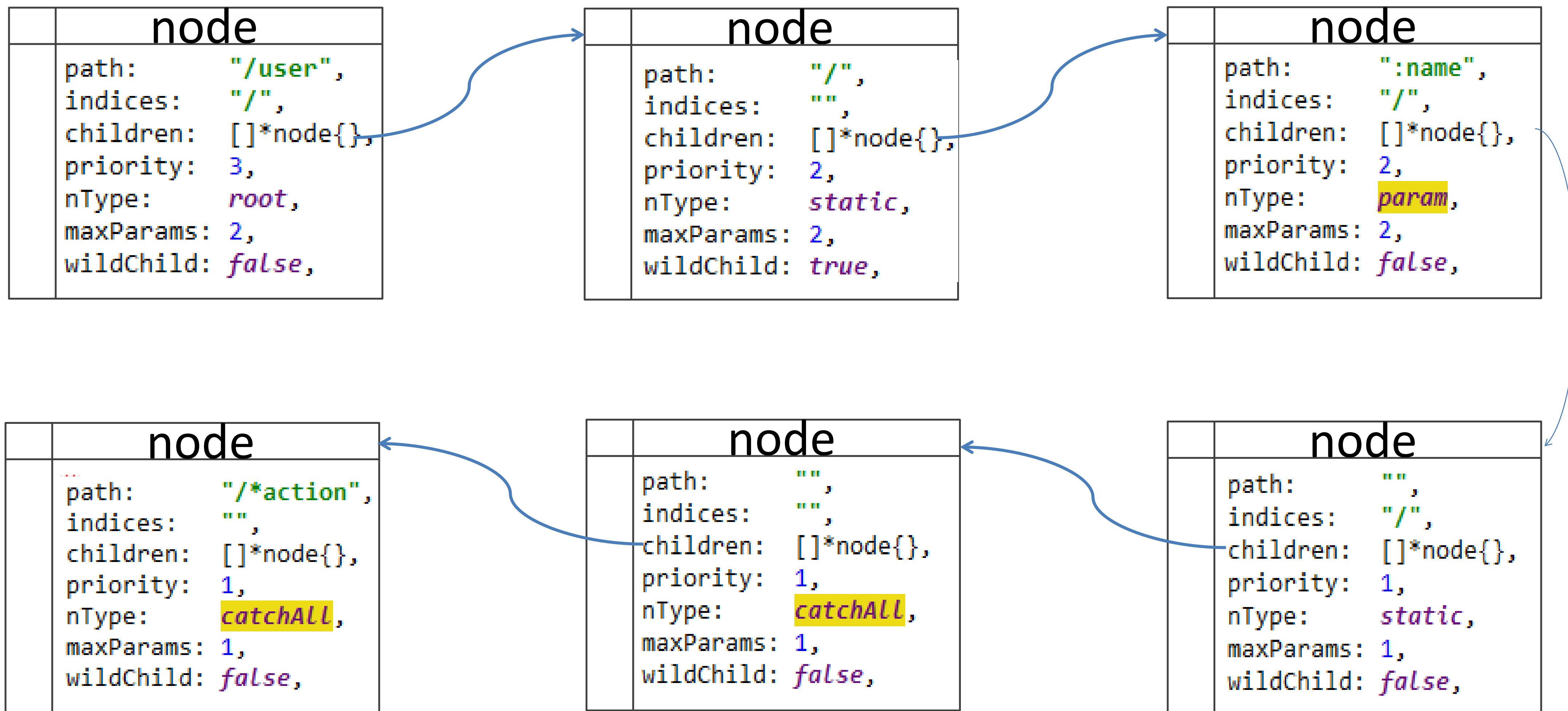
GET /user

GET /user/:name

GET /user/:name/*action



02 | gin框架-路由注册原理





中间件应用

中间件是作用于路由组之上的，不同的路由组可以使用Use函数来添加不同的中间件，单个路由组支持的中间件最大个数不能超过61个。

```
func main() {  
    r := gin.New()  
  
    r.Use(gin.Logger())  
    r.GET(relativePath: "/ping", func(c *gin.Context) {  
        c.String(http.StatusOK, format: "Pong")  
    })  
  
    group := r.Group(relativePath: "/user", gin.Recovery())  
    group.GET(relativePath: "/user/login", func(c *gin.Context) {  
        c.String(http.StatusOK, format: "login success")  
    })  
  
    r.Run()  
}
```



自定义中间件

中间件必需是一个HandlerFunc类型的函数，类型定义如下：

```
type HandlerFunc func(*Context)
```

在中间件处理函数中可以调用Next方法来暂停执行后面的代码，转而执行其它中间件和路由handler，等全部执行完后程序控制权回到当前中间件处理函数中继续运行后面的代码。Next方法并不是中间件处理函数中必须要调用的方法。

如果想中止剩余中间件和路由handler的执行，可调用Abort方法，注意：调用Abort方法后，当前中间件后续的代码会正常执行。



实现log中间件

```
func Logger() gin.HandlerFunc {  
    return func(c *gin.Context) {  
        start := time.Now()  
        path := c.Request.URL.Path  
        raw := c.Request.URL.RawQuery  
  
        // Process request  
        c.Next()  
  
        if raw != "" {  
            path = path + "?" + raw  
        }  
  
        end := time.Now()  
        zap.L().Info( msg: "request info",  
            zap.Duration( key: "duration", end.Sub(start)),  
            zap.String( key: "clientip", c.ClientIP()),  
            zap.String( key: "method", c.Request.Method),  
            zap.Int( key: "status", c.Writer.Status()),  
            zap.String( key: "path", path),  
            zap.String( key: "errmsg", c.Errors.ByType(gin.ErrorTypePrivate).String()),  
            zap.Int( key: "bodysize", c.Writer.Size()),  
        )  
    }  
}
```



Context是gin框架贯穿整个请求上下文结构体，是gin框架的精髓所在，它主要实现了以下几个重要功能。

1、metadata管理

用于在请求处理及各中间件间共享数据

2、请求参数的获取与解析

路由参数和请求参数的获取，支持参数绑定

3、响应处理及渲染

提供多种常见格式的渲染输出



数据共享

Context维护了一个非并发安全的map，用来实现请求作用域内的数据共享，支持存储

任意类型的数据。框架提供下列方法来操作map：

```
Set(key string, value interface{})
Get(key string) (value interface{}, exists bool)
MustGet(key string) interface{}
GetString(key string) (s string)
GetBool(key string) (b bool)
GetInt(key string) (i int)
GetInt64(key string) (i64 int64)
GetFloat64(key string) (f64 float64)
GetTime(key string) (t time.Time)
GetDuration(key string) (d time.Duration)
GetStringSlice(key string) (ss []string)
GetStringMap(key string) (sm map[string]interface{})
GetStringMapString(key string) (sms map[string]string)
GetStringMapStringSlice(key string) (smss map[string][]string)
```




参数绑定

gin支持多种类型的参数绑定，可根据Content-Type头自动选择处理引擎，如：

“application/json” --> JSON binding

“application/xml” --> XML binding

“text/xml” --> XML binding

“multipart/form-data” --> FormMultipart binding



02 | gin框架-Context

```
type InputParam struct {  
    A string  
    B int  
    C float64  
}  
  
func main() {  
    r := gin.Default()  
    // request: curl -X POST -H "Content-Type: application/json"  
    // http://192.168.126.10:8080/json -d '{"a": "1", "b": 1, "c": 1.1}'  
    r.POST(relativePath: "/json", func(c *gin.Context) {  
        var p InputParam  
        _ = c.ShouldBindJSON(&p)  
        // output: {1 1 1.1}  
        fmt.Println(p)  
    })  
  
    r.Run()  
}
```



创建Context

为了减少GC，gin使用pool对象池来管理context，在调用New或Default方法创建engine时设置context的创建函数，如：

```
engine.pool.New = func() interface{} {  
    return engine.allocateContext()  
}  
  
func (engine *Engine) allocateContext() *Context {  
    return &Context{engine: engine}  
}
```



初始化Context

在处理请求前，先从对象池中获取context，然后对context进行初始化，初始化主要设置请求与响应对象，以及重置中间件运行index。请求处理完后放回pool前不会再对context做清理操作。

// 请求处理函数

```
func (engine *Engine) ServeHTTP(w http.ResponseWriter, req *http.Request) {  
    // 从pool中获取context,当pool中不存在对象时会调用new方法创建对象  
    c := engine.pool.Get().(*Context)  
    // 初始化writermem,赋值ResponseWriter对象  
    c.writermem.reset(w)  
    c.Request = req  
    // 重置所有字段,将index置为-1  
    c.reset()  
  
    // 处理请求  
    engine.handleHTTPRequest(c)  
  
    // 将对象放回pool中  
    engine.pool.Put(c)  
}
```

```
func (c *Context) reset() {  
    c.Writer = &c.writermem  
    c.Params = c.Params[0:0]  
    c.handlers = nil  
    c.index = -1  
    c.fullPath = ""  
    c.Keys = nil  
    c.Errors = c.Errors[0:0]  
    c.Accepted = nil  
    c.queryCache = nil  
    c.formCache = nil  
}
```

}

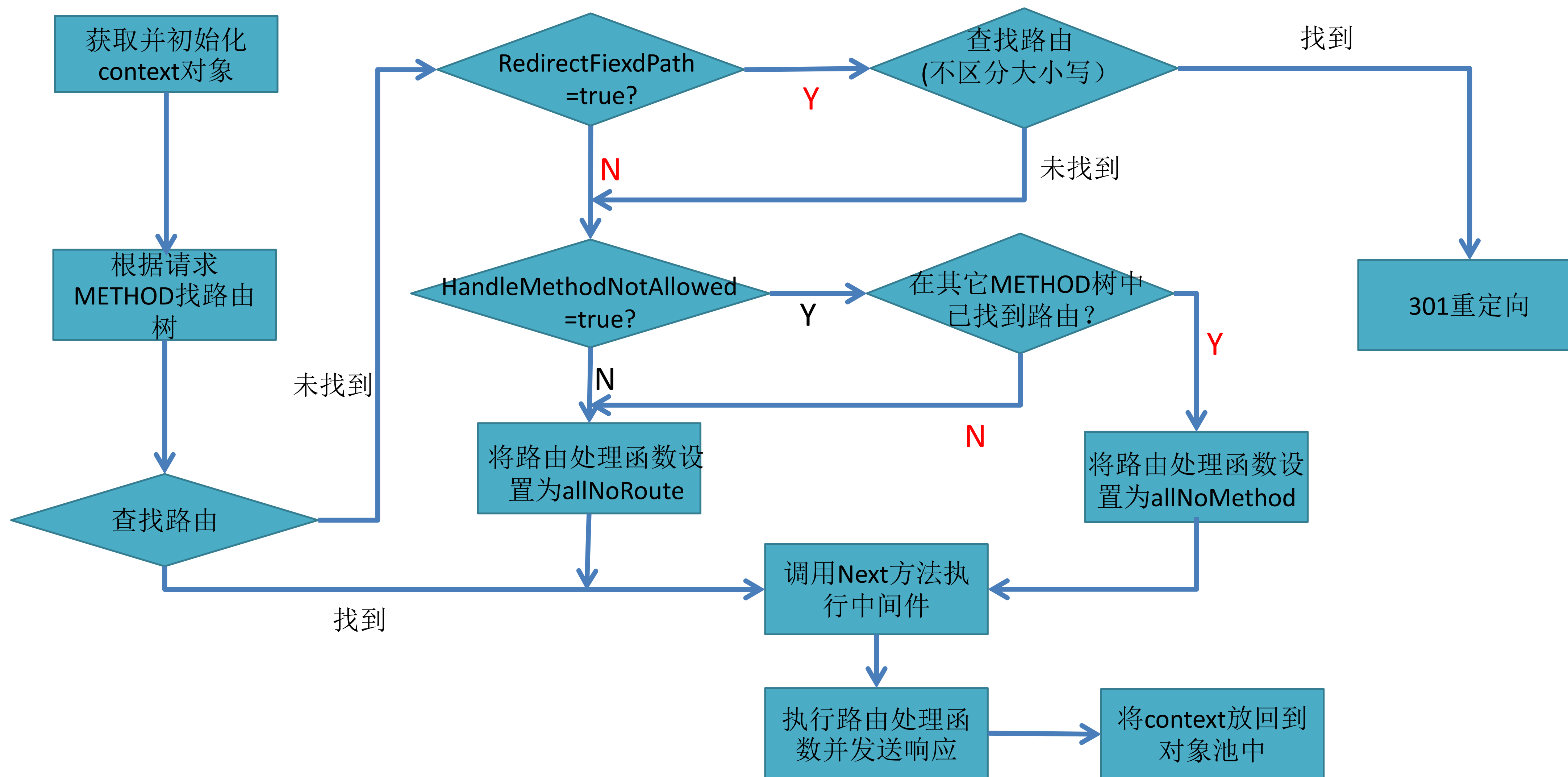


并发问题

对context的操作不是并发安全的，应避免在多个goroutine中访问同一个context。当存在多个goroutine同时访问或在请求作用域外访问时，可使用Copy方法复制一份。



02 | gin框架-请求处理流程





03 常用包篇



问题一

错误信息中没记录堆栈, 无法得知是那行代码返回的错误。如右图所示, 当函数返回错误时, 是第64行返回的, 还是71行或者77行。

```
53 func (db *DB) GetList(where map[string]interface{}, selectFields []string, result interface{}) error {
54     if db.ctx != nil {
55         ctx, exists := trace.GetTraceCtx(db.ctx)
56         if exists {
57             span, _ := opentracing.StartSpanFromContext(ctx, operationName: "GetList")
58             defer span.Finish()
59         }
60     }
61
62     cond, vals, err := builder.BuildSelect(db.Table, where, selectFields)
63     if nil != err {
64         return err
65     }
66     now := time.Now()
67     defer slowLog(fmt.Sprintf(format: "cond(%s) args(%+v)", cond, vals), now)
68
69     rows, err := db.Query(cond, vals...)
70     if err != nil {
71         return err
72     }
73     defer rows.Close()
74
75     err = scanner.Scan(rows, result)
76     if err != nil {
77         return err
78     }
79
80     return nil
81 }
```




问题二

问题1的一个解决方法是在返回错误前输出日志，如右图所示。但这样会导致另一个问题，重复的错误日志过多；GetList函数输出了日志，调用GetList的函数又输出日志，当嵌套层次很多时，会产生大量的重复日志。

```
func (db *DB) GetList(where map[string]interface{}, selectFields []string, result interface{}) error {
    if db.ctx != nil {
        ctx, exists := trace.GetTraceCtx(db.ctx)
        if exists {
            span, _ := opentracing.StartSpanFromContext(ctx, operationName: "GetList")
            defer span.Finish()
        }
    }

    cond, vals, err := builder.BuildSelect(db.Table, where, selectFields)
    if nil != err {
        zap.L().Error(msg: "buildSelect failed", zap.Error(err))
        return err
    }

    now := time.Now()
    defer slowLog(fmt.Sprintf(format: "cond(%s) args(%+v)", cond, vals), now)

    rows, err := db.Query(cond, vals...)
    if err != nil {
        zap.L().Error(msg: "buildSelect failed", zap.Error(err))
        return err
    }
    defer rows.Close()

    err = scanner.Scan(rows, result)
    if err != nil {
        zap.L().Error(msg: "scan failed", zap.Error(err))
        return err
    }

    return nil
}
```



为解决这上述两个问题，可以通过结构体组合来扩充errors的功能，记录函数调用栈。

为避免重复造轮子，可以使用github.com/pkg/errors包，它为我们解决了以上两个问题，并提供了非常多的功能。



pkg/errors

新建一个带调用栈的error:

```
func New(message string) error
```

封装现有的error, 可使用以下三个函数之一:

```
// 仅附加message日志
```

```
func WithMessage(err error, message string) error
```

```
// 仅附加调用栈
```

```
func WithStack(err error) error
```

```
// 附加调用栈和message日志
```

```
func Wrap(err error, message string) error
```

找出产生错误的那个error

```
func Cause(err error) error
```



03 | 错误处理

```
package main

import (
    "fmt"
    "github.com/pkg/errors"
)

func main() {
    err := Foo()
    fmt.Printf(format: "%+v\n\n", errors.Cause(err))
    fmt.Printf(format: "%+v\n", Foo2())
}

func Foo2() error {
    return errors.WithMessage(Foo(),
        | message: "call Foo2")
}

func Foo() error {
    return errors.New(message: "new error")
}
```

```
new error
main.Foo
    /codes/golang/src/modtest-main/main.go:19
main.main
    /codes/golang/src/modtest-main/main.go:9
runtime.main
    /usr/local/gol1.13.6/src/runtime/proc.go:203
runtime.goexit
    /usr/local/gol1.13.6/src/runtime/asm_amd64.s:1357

new error
main.Foo
    /codes/golang/src/modtest-main/main.go:19
main.Foo2
    /codes/golang/src/modtest-main/main.go:15
main.main
    /codes/golang/src/modtest-main/main.go:11
runtime.main
    /usr/local/gol1.13.6/src/runtime/proc.go:203
runtime.goexit
    /usr/local/gol1.13.6/src/runtime/asm_amd64.s:1357
call Foo2
```



03 | json读取

当接收到一个json字符串时，先定义结构体变量，然后调用json.Unmarshal函数将json串解析到变量中，通过访问变量来读取json字段值。当json串包含的字段很多，而我们又仅需要其中几个字段时，如何在不定义结构体的情况下获取需要的字段？



gjson

gjson是一个从json串中读取值的包，并提供数组翻转、移除空白字符等高级功能。

```
package main

import "github.com/tidwall/gjson"

const json = `{"name":{"first":"Janet","last":"Prichard"},"age":47}`

func main() {
    value := gjson.Get(json, path: "name.last")
    // output: Prichard
    println(value.String())
}
```



03 | json读取

键 路 径 语 法

```
const json = `{
  "name": {"first": "Tom", "last": "Anderson"},
  "age": 37,
  "children": ["Sara", "Alex", "Jack"],
  "fav.movie": "Deer Hunter",
  "friends": [
    {"first": "Dale", "last": "Murphy", "age": 44, "nets": ["ig", "fb", "tw"]},
    {"first": "Roger", "last": "Craig", "age": 68, "nets": ["fb", "tw"]},
    {"first": "Jane", "last": "Murphy", "age": 47, "nets": ["ig", "tw"]}
  ]
}
```

```
func main() {
  values := gjson.GetMany(json,
    path...: "name.last", "age", "children", "children.#", "children.1",
    "child*.2", "c?ildren.0", "fav\\.movie", "friends.#.first",
    "friends.1.last")
  for _, value := range values {
    fmt.Println(value)
  }
}
```

```
Anderson
37
["Sara","Alex","Jack"]
3
Alex
Jack
Sara
Deer Hunter
["Dale","Roger","Jane"]
Craig
```



修饰符

@reverse: 翻转数组或对象

@ugly: 移除所有空白字符

@pretty: 美化json串

@this: 返回当前元素

@valid: 验证json串是否有效

@join: 将多个对象合并为一个对象

```
package main

import (
    "fmt"
    "github.com/tidwall/gjson"
)

const json = `{"name": {"first": "Tom", "last": "Anderson"}}`

func main() {
    value := gjson.Get(json, path: "name|@reverse")
    // output: {"last": "Anderson", "first": "Tom"}
    fmt.Println(value)
}
```




sjson

gjson只能用来获取json串值，不能修改json串。跟gjson类似，sjson也不需要解码json串就能修改或删除字段值。

```
package main

import (
    "github.com/tidwall/sjson"
)

const json = `
{
    "name": {"first": "Tom", "last": "Prichard"},
    "age": 37,
    "children": ["Sara", "Alex", "Jack"],
    "fav.movie": "Deer Hunter",
    "friends": [
        {"first": "James", "last": "Murphy"},
        {"first": "Roger", "last": "Craig"}
    ]
}
`

func main() {
    value, _ := sjson.Set(json, path: "name.last", value: "Anderson")
    println(value)

    value, _ = sjson.Delete(json, path: "friends.1")
    println(value)
}
```



当实现配置热加载时，需要实时监控配置文件的变化。`fswatch`是跨平台的可监听目录或文件改动的库。它提供了Create、Write、Remove、Rename、Chmod等事件，当有事件触发时通过channel通知调用方。



03 | 文件监控

```
func FileNotify(dirname string) error {  
    watcher, err := fsnotify.NewWatcher()  
    if err != nil {  
        return err  
    }  
    defer watcher.Close()  
    err = watcher.Add(dirname)  
    if err != nil {  
        return err  
    }  
  
    for {  
        select {  
        case event, ok := <-watcher.Events:  
            if !ok {  
                break  
            }  
  
            fmt.Printf(format: "%s %s\n", event.Name, event.Op)  
            // vim 修改文件时会先触发Create事件，再触发Write事件  
        case err := <-watcher.Errors:  
            fmt.Printf(format: "error: %+v", err)  
        }  
    }  
}
```



golang单元测试，传统的做法是用if语句来判断错误处理，出错时调用Error或Fatal输出错误信息。当测试case太多时，代码中会充满大量的if语句，可使用断言代替if语句来写出更优雅的代码。

stretchr/testify测试框架具有断言、测试套件、mock等功能。



assert&require

两种本质区别是条件未满足时require终止当前测试函数，而assert仅返回bool值。

```
package main

import (
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
    "testing"
)

func TestAssert(t *testing.T) {
    v := 10
    assert.Equal(t, v, actual: 100)
    // 以下语句会被执行
    t.Log(args...: "end")
}

func TestRequire(t *testing.T) {
    v := 10
    require.Equal(t, v, actual: 100)
    // 以下语句不会被执行
    t.Log(args...: "end")
}
```



测试套件

复杂的场景下，单元测试会有先决条件，比如：log组件初始化、连接池初始化等等，单元测试结束后还需主动关闭。testify框架提供了suite包来解决此问题。

右图中的代码，在运行go test命令时首先执行TestMysqlTestSuite函数，调用suite.Run触发单元测试，suite结构体方法的执行顺序：SetupSuite->Test*->TearDownSuite。

```
type MysqlTestSuite struct {
    suite.Suite
    db *DB
}

// 测试前初始化方法
func (suite *MysqlTestSuite) SetupSuite() {
    InitMysql(ModeFile, "motor_test", "default", "test")
    suite.db = GetDB(nil, "testdb", "test", WRITE)
}

func (suite *MysqlTestSuite) TestInsert() {
    _, err := suite.db.Insert()
    require.NoError(suite.T(), err)
}

func (suite *MysqlTestSuite) TestUpdate() {
    _, err := suite.db.Update()
    require.NoError(suite.T(), err)
}

// 测试结束后清理方法
func (suite *MysqlTestSuite) TestDownSuite() {
    suite.db.Close()
}

func TestMysqlSuite(t *testing.T) {
    suite.Run(t, new(MysqlTestSuite))
}
```



mock

testify框架

mock功能的不足

之处是要实

现被mock对象

的所有方法

```
package main

type MessageClient interface {
    Send() error
}

type SMSClient struct{}

func (sms SMSClient) Send() error {
    // todo 发送短信
    return nil
}

type MyService struct {
    msgClient MessageClient
}

func (my *MyService) DoSomething() error {
    // todo 添加业务逻辑
    return my.msgClient.Send()
}

func main() {
    srv := MyService{SMSClient{}}
    srv.DoSomething()
}
```

```
package main

import (
    "github.com/stretchr/testify/mock"
    "testing"
)

type smsServiceMock struct {
    mock.Mock
}

// 实现需要mock的方法
func (m *smsServiceMock) Send() error {
    // 获取第22行Return方法传递的参数
    args := m.Called()
    return args.Error(0)
}

func TestMyService(t *testing.T) {
    smsService := new(smsServiceMock)
    // 调用Send Mock方法时返回nil, 表示发送成功
    smsService.On("Send").Return(nil)

    srv := MyService{smsService}
    srv.DoSomething()

    // 验证mock是否成功
    smsService.AssertExpectations(t)
}
```



04 中间件篇



04 | jwt中间件

JSON Web Token (JWT) 是一个开放的标准 (RFC 7519), 是一种基于 json 的无状态的认证机制, 它由 header、payload、以及 signature 组成。相比 session 方法, 它具有无状态、易扩展、更安全等特点。

我们使用 jwt-go 包以自定义中间件的方式将 jwt 集成到 gin 框架中。



生成与验证token

使用HMAC-SHA356算法来做签名。

```
package jwt

import (
    "fmt"
    "errors"
    "github.com/dgrijalva/jwt-go"
)

type JWT struct {
    secret string
}

type MotorClaims struct {
    jwt.StandardClaims
    Data map[string]interface{}
}

func NewJWT(secret string) *JWT {
    return &JWT{secret}
}

// 生成token
func (j *JWT) GenToken(claims *MotorClaims) (string, error) {
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return token.SignedString([]byte(j.secret))
}
```



04 | jwt中间件

token验证失败时
注意区分是token
无效还是已过期。

```
// 验证token
func (j *JWT) ParseToken(tokenStr string) (*MotorClaims, error) {
    token, err := jwt.ParseWithClaims(tokenStr, &MotorClaims{}, func(token *jwt.Token) (interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("Unexpected signing method: %v", token.Header["alg"])
        }

        return []byte(j.secret), nil
    })
    if err != nil {
        return nil, err
    }

    if claims, ok := token.Claims.(*MotorClaims); ok && token.Valid {
        return claims, nil
    }

    return nil, errors.New("无效的token")
}

// 判断token是否过期
func (j *JWT) IsExpires(err error) bool {
    if ve, ok := err.(*jwt.ValidationError); ok {
        if ve.Errors&jwt.ValidationErrorExpired != 0 {
            return true
        }
    }

    return false
}
```



中间件

```
9 func Jwt(secret string) gin.HandlerFunc {
10     j := jwt.NewJWT(secret)
11     return func(c *gin.Context) {
12         tokenStr := c.Request.Header.Get("JWT-TOKEN")
13         if tokenStr == "" {
14             c.JSON(http.StatusOK, gin.H{
15                 "status": -1,
16                 "errmsg": "请求未携带jwt-token",
17                 "data": nil,
18             })
19             c.Abort()
20             return
21         }
22     }
23 }
```

检查JWT-TOKEN http头中是否有token，如果没有客户端需发起认证，如果有就验证token是否有效，并将凭证存储到gin.Context中。

```
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
claims, err := j.ParseToken(tokenStr)
if err != nil {
    if j.IsExpires(err) {
        c.JSON(http.StatusOK, gin.H{
            "status": 1,
            "errmsg": "token已过期",
            "data": nil,
        })
        c.Abort()
        return
    }

    c.JSON(http.StatusOK, gin.H{
        "status": 1,
        "errmsg": "token无效",
        "data": nil,
    })
    c.Abort()
    return
}

c.Set("jwtClaims", claims)
}
```



04 | 请求Log中间件

gin框架自带的log中间件输出信息太少，格式不通用。我们需要实现一个自定义的log中间件，将请求日志格式化/nginx的accesslog风格，并记录请求处理的错误信息以及请求耗时。

```
'$remote_addr - - [$time_local] "$request" $status $body_bytes_sent  
"$http_referer" "$http_user_agent";
```



04 | 请求Log中间件

```
12 func Logger() gin.HandlerFunc {
13     return func(c *gin.Context) {
14         start := time.Now()
15         path := c.Request.URL.Path
16         raw := c.Request.URL.RawQuery
17
18         // Process request
19         c.Next()
20
21         if raw != "" {
22             path = path + "?" + raw
23         }
24
25         end := time.Now()
26         local, _ := time.LoadLocation("Local")
27         referer := c.Request.Header.Get("Referer")
28         if referer == "" {
29             referer = "-"
30         }
31         accessLog := fmt.Sprintf("%s - - [%s] \"%s %s %s\" %d %d \"%s\" \"%s\"",
32             c.ClientIP(),
33             end.In(local).Format("02/Jan/2006:15:04:05"),
34             c.Request.Method,
35             c.Request.URL.Path,
36             c.Request.Proto,
37             c.Writer.Status(),
38             c.Writer.Size(),
39             referer,
40             c.Request.Header.Get("User-Agent"),
41
43         zap.L().Info(accessLog,
44             zap.Duration("latency", end.Sub(start)),
45             zap.String("path", path),
46             zap.String("errmsg", c.Errors.ByType(gin.ErrorTypePrivate).String()),
47         )
48     }
49 }
```

注意：使用Format格式化日期时间时一定要用"2006-01-02 15:04:05" 在处理业务过程中可以将错误记录到context中，第46行将业务错误与请求日志关联起来。



04 | metrics中间件

metrics是指可聚合的数据，当为线上服务做各项指标实时监控、统计时我们就需要用到它，比如：qps统计、请求延迟矩阵、请求错误统计等等。

Prometheus是golang生态中一个很重要的开源监控系统，也是第二个加入CNCF基金会的开源项目。它提供了如下四种基本metric类型供我们使用：

Counter: 类似计数器，常用于累计值，如：请求次数，错误次数等，值一直增加，不会减少。

Gauge: 仪表盘，值可增可减，常用于描述反应系统当前状态的指标，如：内存使用情况等。

Histogram: 柱状图，常用于跟踪事件发生的规模，分析数据样本，并可对内容进行分组，如：响应时间100ms内的请求次数，大于100ms的请求次数等。

Summary: 此类型的测量对象跟histogram类似，但两者有着很大的差别，summary每次在调用Observe时会计算百分位数，会增加客户端的性能消耗，而histogram在服务端做的聚合。其次在客户端做聚合以及计算百分位后，无法精确的对整个集群计算百分位数。



Counter

```
var ReqCnt = prometheus.NewCounterVec(  
    prometheus.CounterOpts{  
        Namespace: "test",  
        Name:      "requests_total",  
        Help:      "http client requests count",  
    }, []string{"path", "code"})  
  
func init() {  
    prometheus.MustRegister(ReqCnt)  
}  
  
func main() {  
    r := gin.Default()  
    r.GET("/counter/*action", func(c *gin.Context) {  
        ReqCnt.WithLabelValues(c.Request.URL.Path,  
            http.StatusText(http.StatusOK)).Inc()  
    })  
    r.GET("/metrics", gin.WrapH(promhttp.Handler()))  
    r.Run()  
}
```

右图注册了一个Counter类型，分别按uri和响应状态码来统计请求次数。

发送用户请求：

<http://127.0.0.1:8080/counter/1>

<http://127.0.0.1:8080/counter/2>

获取metrics数据：

<http://127.0.0.1:8080/metrics>

```
# HELP test_requests_total http client requests count  
# TYPE test_requests_total counter  
test_requests_total{code="OK",path="/counter/1"} 1  
test_requests_total{code="OK",path="/counter/2"} 1
```




04 | metrics中间件

Gauge

```
var memGauge = prometheus.NewGaugeVec(  
    prometheus.GaugeOpts{  
        Namespace: "test",  
        Name: "memory_usage",  
    },  
    []string{"name"},  
)  
  
func init() {  
    prometheus.MustRegister(memGauge)  
}  
  
func main() {  
    r := gin.Default()  
    r.GET("/memusage", func(c *gin.Context) {  
        var ms runtime.MemStats  
        runtime.ReadMemStats(&ms)  
        memGauge.WithLabelValues("name").Set(float64(ms.HeapAlloc))  
    })  
    r.GET("/metrics", gin.WrapH(promhttp.Handler()))  
    r.Run()  
}
```

右图注册了一个Gauge类型的metrics来记录内存使用情况，内存占用量每次记录时可增可减。

发送用户请求：

<http://127.0.0.1:8080/memusage>

获取metrics数据：

<http://127.0.0.1:8080/metrics>

```
# HELP test_memory_usage  
# TYPE test_memory_usage gauge  
test_memory_usage{name="name"} 1.589736e+06
```



Histogram

```
var ReqDur = prometheus.NewHistogramVec(  
    prometheus.HistogramOpts{  
        Namespace: "test",  
        Name:      "duration",  
        Buckets:   []float64{0.5, 1, 2, 5, 10},  
        Help:      "http client requests duration",  
    }, []string{"path"})  
  
func init() {  
    prometheus.MustRegister(ReqDur)  
}  
  
func main() {  
    r := gin.Default()  
    r.GET("/reqdur", func(c *gin.Context) {  
        rand.Seed(time.Now().Unix())  
        ReqDur.WithLabelValues(c.Request.URL.Path).  
            Observe(float64(rand.Intn(15)))  
    })  
    r.GET("/metrics", gin.WrapH(promhttp.Handler()))  
    r.Run()  
}
```

右图定义了5个buckets，分别表示请求耗时范围为：

0~0.5s、0.5~1s、1~2s、2~5s、5~10s。

发送5次用户请求：

<http://127.0.0.1:8080/reqdur>

获取metrics数据：

<http://127.0.0.1:8080/metrics>

```
# HELP test_duration http client requests duration  
# TYPE test_duration histogram  
test_duration_bucket{path="/reqdur",le="0.5"} 0  
test_duration_bucket{path="/reqdur",le="1"} 0  
test_duration_bucket{path="/reqdur",le="2"} 0  
test_duration_bucket{path="/reqdur",le="5"} 0  
test_duration_bucket{path="/reqdur",le="10"} 2  
test_duration_bucket{path="/reqdur",le="+Inf"} 5  
test_duration_sum{path="/reqdur"} 53  
test_duration_count{path="/reqdur"} 5
```

duration_count:

记录请求次数。

duration_sum:记

录总耗时。



04 | metrics中间件

Summary

```
var ReqDur = prometheus.NewSummaryVec(
    prometheus.SummaryOpts{
        Namespace: "test",
        Name:       "duration",
        Objectives: map[float64]float64{
            0.5: 0.05,
            0.95: 0.01,
            0.99: 0.001,
        },
        Help: "http client requests duration",
    },
    []string{"path"},
)

func init() {
    prometheus.MustRegister(ReqDur)
}

func main() {
    r := gin.Default()
    r.GET("/reqdur", func(c *gin.Context) {
        rand.Seed(time.Now().Unix())
        ReqDur.WithLabelValues(c.Request.URL.Path).
            Observe(float64(rand.Intn(15)))
    })
    r.GET("/metrics", gin.WrapH(promhttp.Handler()))
    r.Run()
}
```

Objectives定义了样本值的分位数分布详情，0.5:0.05表示中位数的误差值是0.05，值范围： $[0.5-0.05, 0.5+0.05]$ 。

获取metrics数据：

<http://127.0.0.1:8080/metrics>

```
# HELP test_duration http client requests duration
# TYPE test_duration summary
test_duration{path="/reqdur",quantile="0.5"} 6
test_duration{path="/reqdur",quantile="0.95"} 7
test_duration{path="/reqdur",quantile="0.99"} 7
test_duration_sum{path="/reqdur"} 27
test_duration_count{path="/reqdur"} 5
```

5次请求的耗时为：5、7、6、6、3。

中位数值为6，95分位数值为7。



中间件

```
func Init(namespace string) {
    ReqCnt = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Namespace: namespace,
            Name:      "requests_total",
            Help:      "http client requests count",
        }, []string{"path", "code"})

    ReqDur = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Namespace: namespace,
            Name:      "duration_ms",
            Buckets:   []float64{0.5, 1, 2, 5, 10},
            Help:      "http client requests duration(n
        }, []string{"path"})

    ReqErr = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Namespace: namespace,
            Name:      "requests_errcode_total",
            Help:      "http client error requests cour
        }, []string{"path", "code"})

    prometheus.MustRegister(ReqCnt, ReqDur, ReqErr)
```

```
}func Metrics() gin.HandlerFunc {
    return func(c *gin.Context) {
        if c.Request.URL.Path == metrics.DefaultPath {
            c.Next()
            return
        }

        start := time.Now()

        c.Next()

        status := c.Writer.Status()
        sstatus := strconv.Itoa(status)
        path := c.Request.URL.Path

        metrics.ReqCnt.WithLabelValues(path, sstatus).Inc()
        metrics.ReqDur.WithLabelValues(path).
            Observe(float64(time.Since(start).Milliseconds()))
        if status >= http.StatusInternalServerError {
            metrics.ReqErr.WithLabelValues(path, sstatus).Inc()
        }
    }
}
```



微服务架构中，单个请求往往会涉及到多个模块，或多个服务。当请求耗时久时我们需要快速确定请求背后调用了那些服务，是那个服务导致请求耗时长。分布式追踪链就是为解决此类问题而诞生的，它可以记录每个服务、每个模块，甚至时每个函数的处理时长。

OpenTracing是一种分布式追踪的API规范，它是第三个加入CNCF的项目，它提供统一的接口，方便我们开发者在项目中集成一种或多种分布式追踪系统的实现。目前基于OpenTracing规范并比较流行的分布式追踪系统有Jaeger和Zipkin，前者是Uber开源的项目，后者是Twitter分司贡献的开源项目。这里我们选择Jaeger，将其集成中gin框架中。



04 | tracing中间件

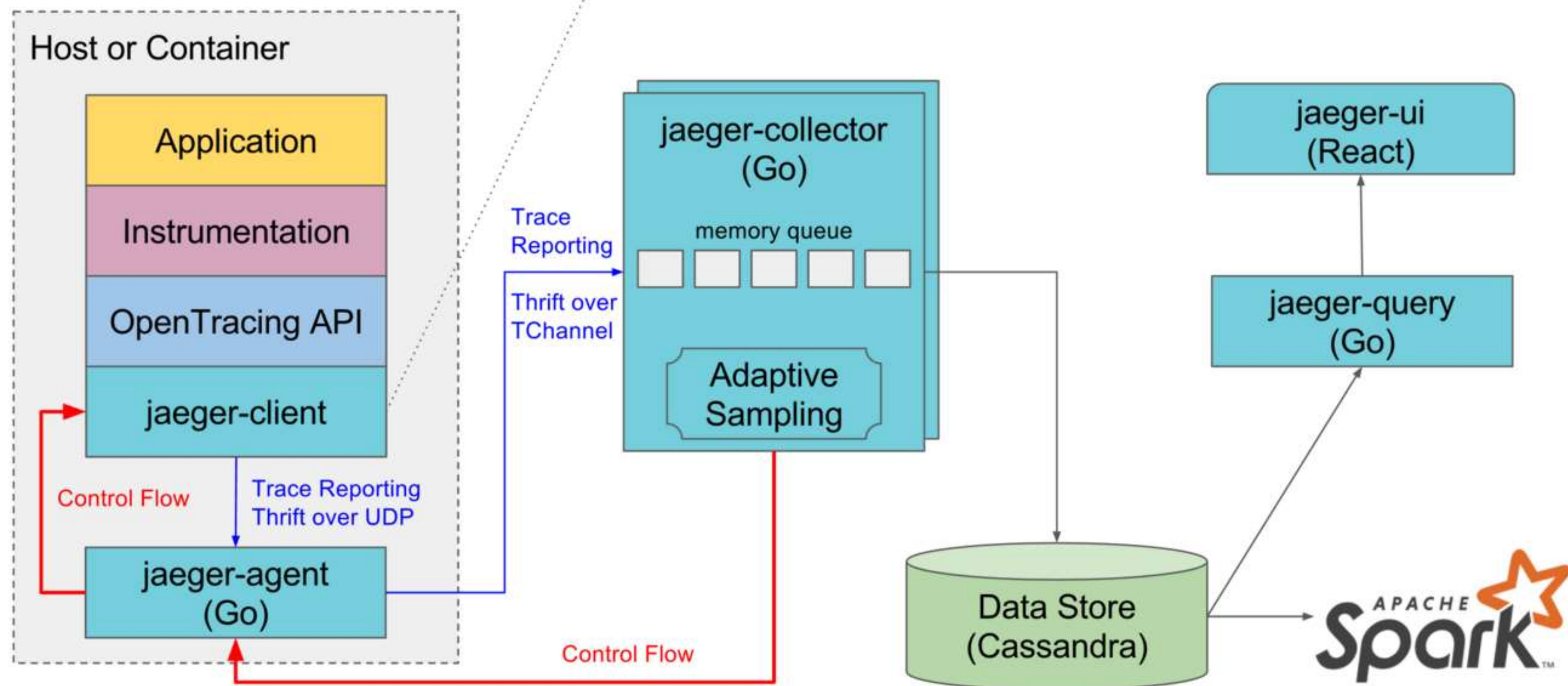
Jaeger架构

jaeger-client: jaeger客户端，实现了opentracing的接口。负责生成及发送span给agent或collector。

jaeger-agent: 缓存client发来的span，并批量发送给collector，是可选的组件。

jaeger-collector: 接收agent或client发来的span，并写入持久化存储中。

JAEGER





数据模型

每个调用链包含多个span，一个span可以理解为包含埋点信息的结点，span之间的关系称为References，分别有ChildOf和FollowsFrom两种References。

一个span通常包含以下状态：

- 操作名
- 起始时间
- 结束时间
- Tags (一个或多个KV值)
- Logs (一个或多个带时间的KV值)
- 一个SpanContext
- References

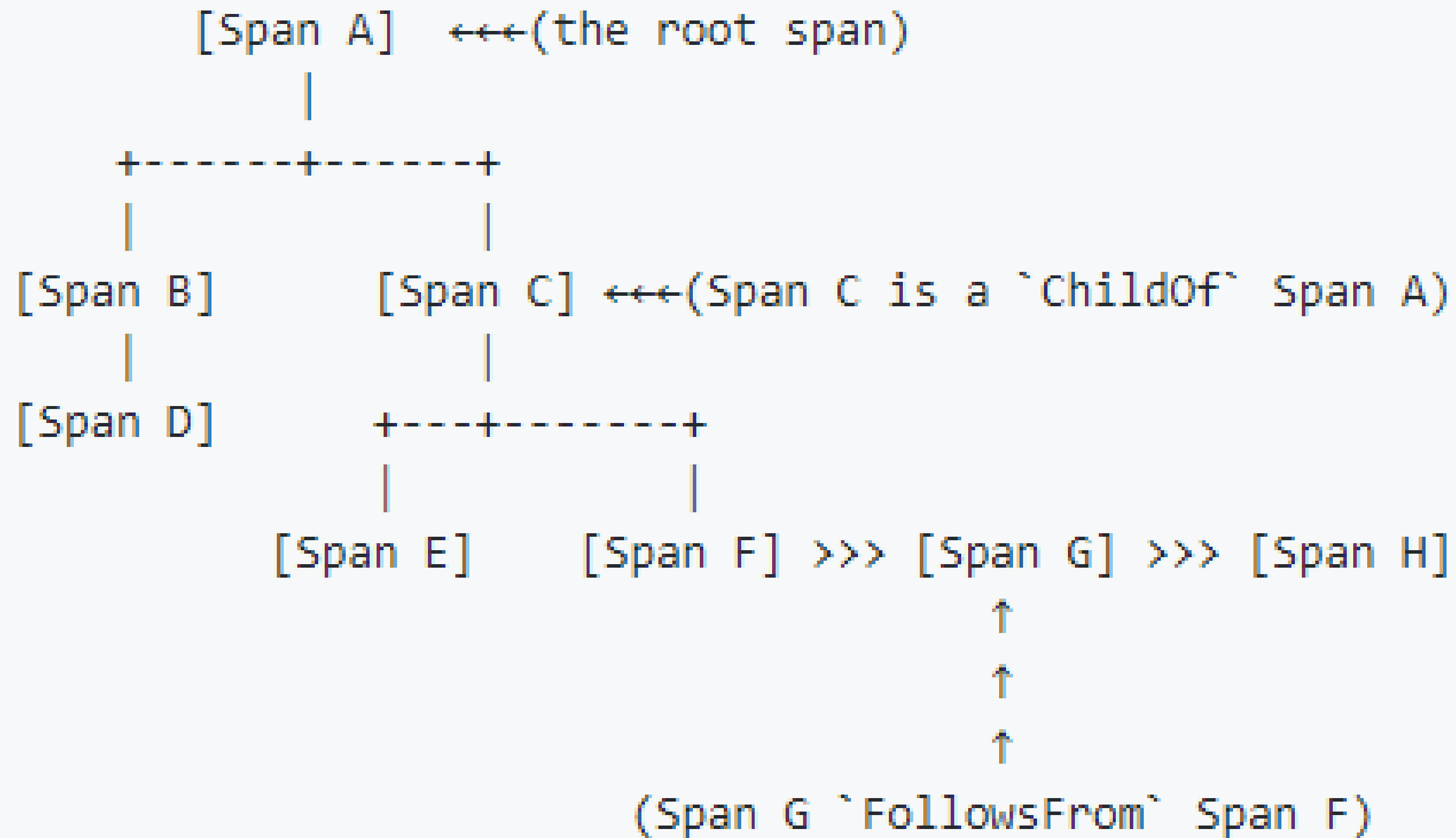
spanContext包含以下状态：

- TraceID
- Baggage Items (跨进程传输的数据)



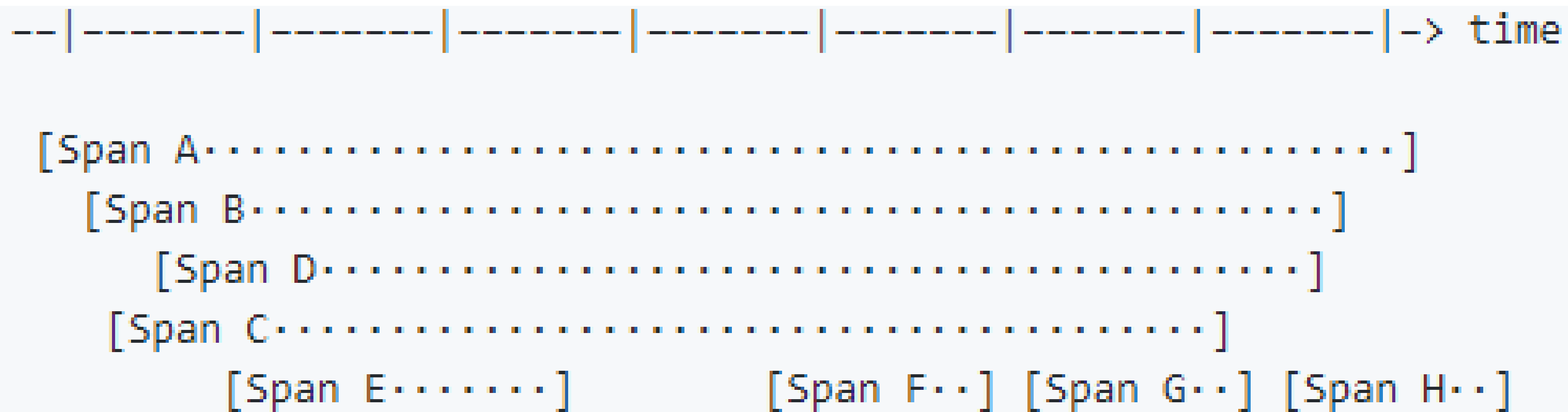
ChildOf

Causal relationships between Spans in a single Trace





FollowsFrom



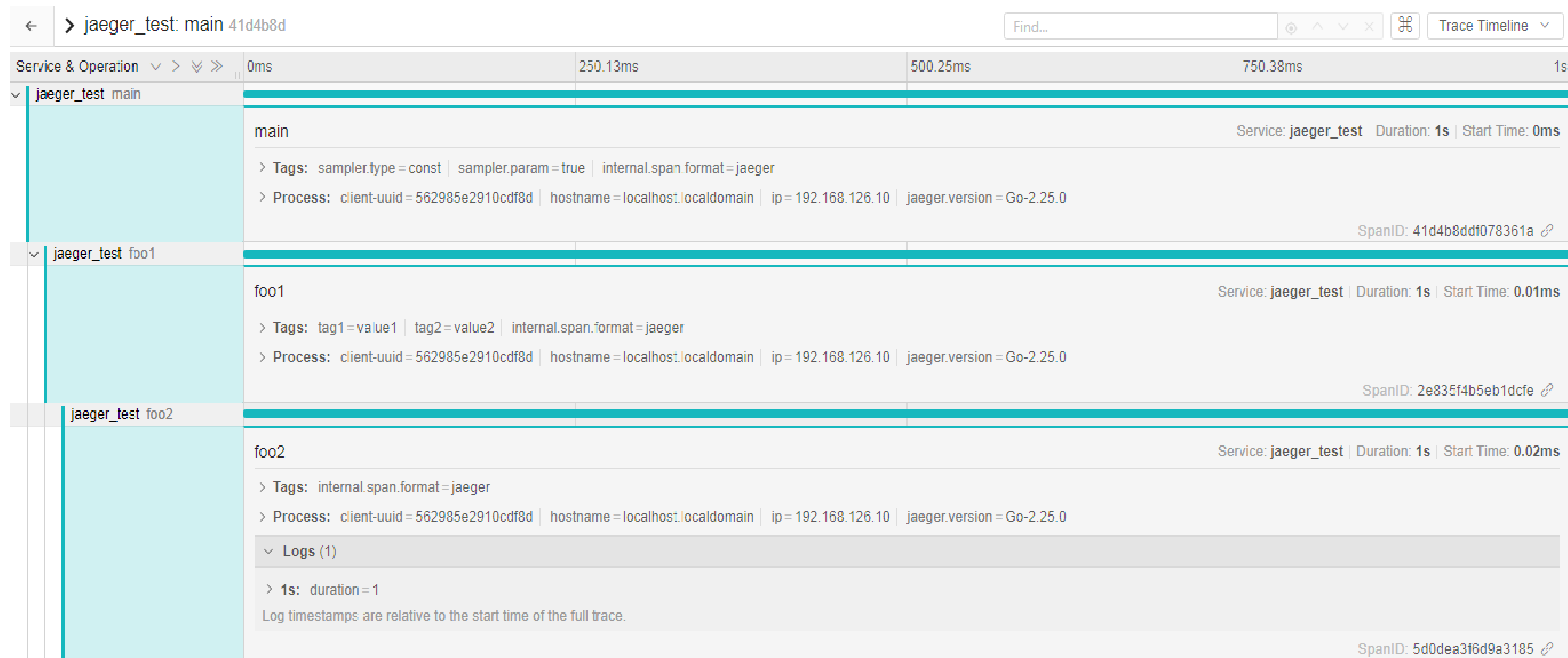


单进程实例

```
func main() {  
    // root span  
    rootSp := opentracing.StartSpan("main")  
  
    foo1(rootSp)  
    rootSp.Finish()  
    gCloser.Close()  
}  
  
func foo1(parentSp opentracing.Span) {  
    // childOf references  
    sp := opentracing.StartSpan("foo1", opentracing.ChildOf(parentSp.Context()))  
    defer sp.Finish()  
  
    sp.SetTag("tag1", "value1")  
    sp.SetTag("tag2", "value2")  
    foo2(sp)  
}  
  
func foo2(parentSp opentracing.Span) {  
    sp := opentracing.StartSpan("foo2", opentracing.ChildOf(parentSp.Context()))  
    defer sp.Finish()  
  
    time.Sleep(time.Second)  
    sp.LogFields(log.Int("duration", 1))  
}
```



04 | tracing中间件





跨进程实例Client

```
func main() {  
    span := opentracing.StartSpan("client")  
    span.SetBaggageItem("x", "y")  
    httpClient := &http.Client{}  
    httpReq, _ := http.NewRequest("GET", "http://192.168.126.10:8080/trace", nil)  
  
    // 将trace信息存放到http头中  
    opentracing.GlobalTracer().Inject(  
        span.Context(),  
        opentracing.HTTPHeaders,  
        opentracing.HTTPHeadersCarrier(httpReq.Header))  
    httpClient.Do(httpReq)  
  
    span.Finish()  
    gCloser.Close()  
}
```



跨进程实例Server

```
func main() {  
    r := gin.New()  
  
    r.GET("/trace", func(c *gin.Context) {  
        tracer := opentracing.GlobalTracer()  
        spanCtx, err := tracer.Extract(opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(c.Request.Header))  
        var span opentracing.Span  
        if err != nil {  
            span = tracer.StartSpan(c.Request.URL.Path)  
        } else {  
            span = tracer.StartSpan(c.Request.URL.Path, opentracing.ChildOf(spanCtx))  
        }  
        defer span.Finish()  
  
        // 获取client传过来的数据  
        span.SetTag("baggage_x", span.BaggageItem("x"))  
        c.String(http.StatusOK, "success")  
    })  
  
    r.Run()  
}
```



04 | tracing中间件

Service & Operation	0ms	0.74ms
jaeger_test main	main	
	<p>> Tags: sampler.type = const sampler.param = true internal.span.format = jaeger</p> <p>> Process: client-uuid = 198687871b16a541 hostname = localhost.localdomain </p> <p>Logs (1)</p> <p>> 0.02ms: event = baggage key = x value = y</p> <p>Log timestamps are relative to the start time of the full trace.</p>	
jaeger_test /trace	/trace	
	<p>> Tags: baggage_x = y internal.span.format = jaeger</p> <p>> Process: client-uuid = 42b6f92071521d3d hostname = localhost.localdomain </p>	

GET /trace HTTP/1.1

Host: 192.168.126.10:8080

User-Agent: Go-http-client/1.1

Uber-Trace-Id:

5cd0ff5ed1afdc05:5cd0ff5ed1afdc05:0000000000000000:1

Uberctx-X: y

Accept-Encoding: gzip

HTTP/1.1 200 OK

Content-Type: text/plain; charset=utf-8

Date: Tue, 29 Dec 2020 08:14:15 GMT

Content-Length: 7

success



04 | tracing中间件

gin中间件

首先从http Header中获取Baggage信息，并将其注入到carrier变量中，然后将spanContext和carrier变量保存到gin.Context中。

```
func Trace() gin.HandlerFunc {
    return func(c *gin.Context) {
        if opentracing.IsGlobalTracerRegistered() {
            tracer := opentracing.GlobalTracer()
            spanCtx, err := tracer.Extract(opentracing.HTTPHeaders, opentracing.HTTPHeadersCarrier(c.Request.Header))
            var span opentracing.Span
            if err != nil {
                span = tracer.StartSpan(c.Request.URL.Path)
            } else {
                span = tracer.StartSpan(c.Request.URL.Path, opentracing.ChildOf(spanCtx))
            }
            defer span.Finish()

            carrier := opentracing.TextMapCarrier{}
            // 将请求的Baggage数据inject到carrier中
            err = tracer.Inject(span.Context(), opentracing.TextMap, carrier)
            if err != nil {
                zap.L().Error("tracer.Inject failed",
                    zap.String("clientId", c.ClientIP()),
                    zap.String("url", c.Request.URL.Path),
                    zap.String("method", c.Request.Method),
                    zap.String("errmsg", fmt.Sprintf("%v", err)),
                )
            }

            ctx := opentracing.ContextWithSpan(context.Background(), span)
            ctx = metadata.NewContext(ctx, metadata.Metadata(carrier))
            // 存放到gin框架的context中，方便后面的中间件及路由handler获取
            c.Set(trace.Tracer_Ctx_Key, ctx)
        }
        c.Next()
    }
}
```



04 | tracing中间件

路由handler中可以使用
GetTraceCtx函数获取中间
件注入的spanContext，通
过将此context做为父级
context来创建新的span。
GetTraceID函数用于获取每
个请求的traceid，用于跟
日志绑定。

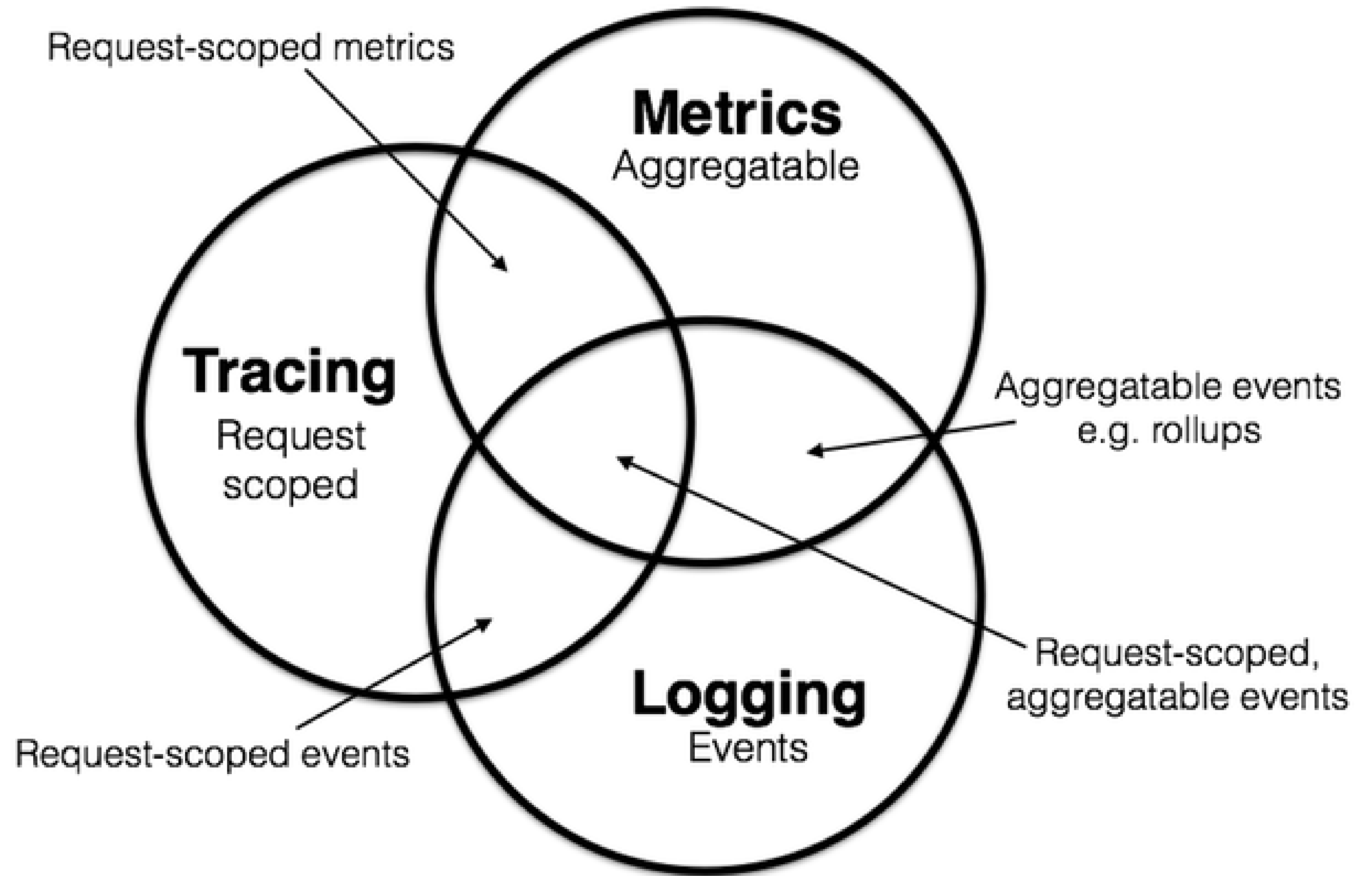
```
func GetTraceCtx(c *gin.Context) (context.Context, bool) {  
    ctx, exists := c.Get(Tracer_Ctx_Key)  
    if !exists {  
        return nil, false  
    }  
    ctx2, ok := ctx.(context.Context)  
    if !ok {  
        return nil, false  
    }  
  
    return ctx2, true  
}  
  
func GetTraceID(span opentracing.Span) (id string, ok bool) {  
    if sc, ok := span.Context().(jaeger.SpanContext); ok {  
        return sc.TraceID().String(), true  
    }  
  
    return "", false  
}
```


04 | log&metrics&tracing的关系

Log: 常用于记录离散的事件，如：记录由某行代码引起的错误或调试信息。

Metrics: 侧重聚合，将离散的数据信息聚合成逻辑意义上的数据。

Tracing: 记录请求处理流程相关信息。





05 应用篇



热加载配置不需要重启服务进程，一般具有以下功能：

- 监控配置文件改动，并加载配置文件。
- 支持watch，业务方实时获取最新配置。
- 并发安全的获取配置。
- 支持常用配置文件格式。

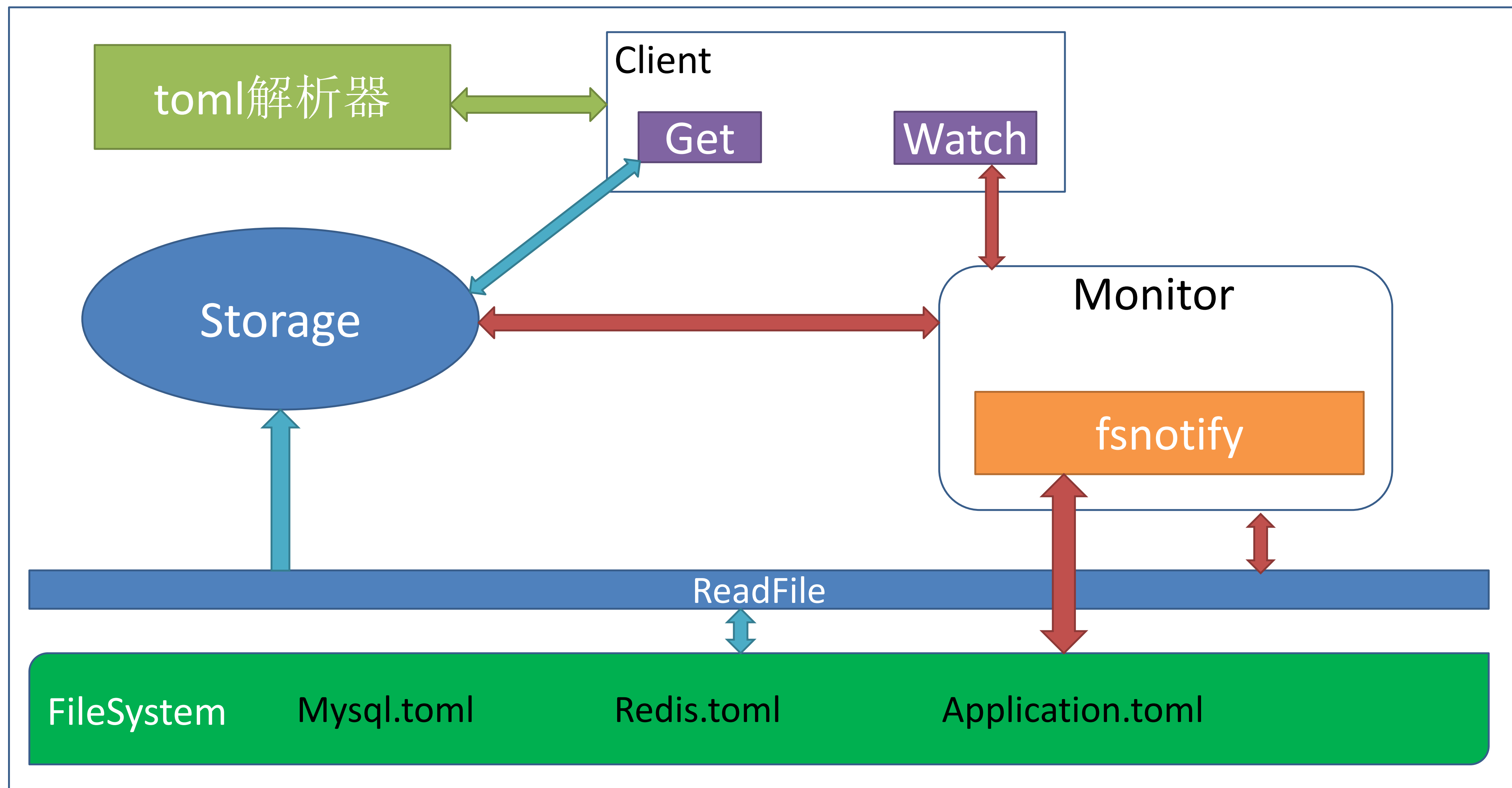


配置文件格式

常用的配置文件格式有json、yaml和toml，json的优势是简单，但不便于描述复杂的配置；yaml的规范太多，且可读性差较。这里我们选择toml，它是一种极简的配置文件格式，相比json和yaml可读性更好，并支持丰富的数据类型，如：float、slice、map等，在描述多重嵌套的配置时非常方便，也不会出现像yaml那样由空格导致的各种问题。



05 | 配置热加载





Storage

storage用于存储配置文件内容，我们将storage结构体定义如下：

```
type Storage struct {  
    values atomic.Value  
}
```

用atomic.Value来存储配置内容，实现配置原子获取。atomic.Value类型是一个空接口，因此可以存储任意类型的值。



Storage

Store方法的入参是一个map，key是配置文件名，value是字符串格式的文件内容。

可通过以下代码获取配置文件内容：

```
conf.Get("application.toml")
```

```
type Storage struct {  
    values atomic.Value  
}  
  
func (p *Storage) Store(s map[string]string) {  
    m := make(map[string]string, len(s))  
    for k, v := range s {  
        m[KeyName(k)] = v  
    }  
  
    p.values.Store(m)  
}  
  
func (p *Storage) Get(key string) string {  
    return p.Load()[KeyName(key)]  
}  
  
func (p *Storage) Load() map[string]string {  
    src := p.values.Load().(map[string]string)  
    dst := make(map[string]string)  
    for k, v := range src {  
        dst[k] = v  
    }  
  
    return dst  
}
```



Watch

每个调用方可以注册监听多个配置的变化，同一个配置也有可能被多方监听，我们使用一个map结构来存储每个配置的监听方，如右图watchChs变量。调用方调用WatchEvent函数注册监听。

```
const (  
    EventTypeUpdate EventType = iota  
)  
type EventType int  
type Event struct {  
    Op EventType  
    Key string  
    Val string  
}  
  
var elock sync.Locker  
var watchChs map[string][]chan Event  
func WatchEvent(keys ...string) <-chan Event {  
    ch := make(chan Event, 1)  
    elock.Lock()  
    defer elock.Unlock()  
    for _, key := range keys {  
        watchChs[key] = append(watchChs[key], ch)  
    }  
  
    // 业务方通过读此channel得知配置有更新  
    return ch  
}
```




Monitor

使用fsnotify第三方包来监控配置文件的改变，当监控到文件被修改时，重新读取文件，并将配置文件写入调用方监听的channel。

```
func Monitor(cpath string) {  
    watcher, err := fsnotify.NewWatcher()  
    if err != nil {  
        log.Printf("fsnotify.NewWatcher failed, err: %+v", err)  
        return  
    }  
    defer watcher.Close()  
    err = watcher.Add(cpath)  
    if err != nil {  
        log.Printf("watcher.Add(%s) failed, err: %+v", cpath, err)  
        return  
    }  
  
    for {  
        select {  
        case event, ok := <-watcher.Events:  
            if !ok {  
                break  
            }  
            // vim 修改文件时会先触发Create事件，再触发Write事件  
            if event.Op&fsnotify.Write != 0 {  
                reloadFile(event.Name)  
            } else {  
                log.Printf("monitor: unsupport event %+v", event)  
            }  
        case err := <-watcher.Errors:  
            log.Printf("error: %+v", err)  
        }  
    }  
}
```



Monitor

```
// read the file again, and send event notification
func reloadFile(name string) {
    // 这里休眠一段时间，避免文件内容还未更新
    time.Sleep(500 * time.Millisecond)
    key := filepath.Base(name)
    val, err := readFile(name)
    if err != nil {
        log.Printf("readFile(%s) failed, error: %+v", name, err)
        return
    }
    m := map[string]string{
        key: val,
    }
    storage.Store(m)

    elock.Lock()
    chs := watchChs[key]
    elock.Unlock()

    for _, ch := range chs {
        select {
        case ch <- Event{Op: EventUpdate, Key: key, Val: val}:
        default:
            log.Printf("event channel full discard file %s update event, content:%+v",
                name, val)
        }
    }
}
```



toml解析器

从storage中获取的配置是字符串格式的，需要将它解析到结构体变量中。假设log日志文件内容如下图所示，右图为解析toml格式配置的代码。

```
# 日志级别
level = "debug"
# 日志格式
encoding = "json"
# 日志文件路径名
outputPaths = ["/tmp/motor/log/application.log"]
# 错误日志文件路径名
errOutputPaths = ["/tmp/motor/log/application.err.log"]
```

```
type LogConf struct {
    Level          string `toml:level`
    Encoding       string `toml:encoding`
    OutputPaths    []string `toml:outputPaths`
    ErrOutputPaths []string `toml:errOutputPaths`
}

func ParseConf() {
    var cfg LogConf
    content := storage.Get("db.toml")
    if content == "" {
        panic("invalid config content")
    }
    err := toml.Unmarshal([]byte(content), &cfg)
    if err != nil {
        panic(err)
    }

    Watch("db.toml")
}

func Watch(key string) {
    go func() {
        var cfg LogConf
        for event := range WatchEvent(key) {
            err := toml.Unmarshal([]byte(event.Val), &cfg)
            if err != nil {
                log.Printf("toml.Unmarshal failed, err:%v", err)
                continue
            }
            // todo
        }
    }()
}
```



toml解析器

假设有如下配置项：

maxAllowedRtMs = "500ms"

此配置项值为一个时间值，如：500ms、1s、5s等。为避免调用方拿到字符串后再去转换成duration。配置框架可通过实现encoding.TextUnmarshaler接口在配置解析时自动将字符串转换为duration。

```
type Duration time.Duration
func (d *Duration) UnmarshalText(text []byte) error {
    value, err := time.ParseDuration(string(text))
    if err == nil {
        *d = Duration(value)
        return nil
    }

    return err
}

type sentinelConf struct {
    MaxAllowedRtMs Duration `toml:"maxAllowedRtMs"`
}

func TestUnmarshalText(t *testing.T) {
    var cfg sentinelConf
    src1 := `maxAllowedRtMs="500ms"`
    _ = toml.Unmarshal([]byte(src1), &cfg)
    assert.Equal(t, time.Duration(cfg.MaxAllowedRtMs),
        500*time.Millisecond)

    src2 := `maxAllowedRtMs="1s"`
    _ = toml.Unmarshal([]byte(src2), &cfg)
    assert.Equal(t, time.Duration(cfg.MaxAllowedRtMs),
        time.Second)
}
```



05 | 连接池

连接池是一种以空间换时间的手段，它负责连接的建立与销毁，并将连接存储在内存中。golang中可使用多种方式来实现连接池，如slice、队列等。golang内置的database/sql包使用slice，而redigo包使用的是队列。

我们以database/sql为例，讲解下它的原理，它有以下三个重要的参数：

MaxOpenConns：池中最大的连接数

MaxIdleConns：最大的空闲连接数

ConnMaxLifetime：空闲连接最大的生命周期，过期后连接被销毁



获取连接

database/sql包调用query或exec方法获取连接分为三个步骤：

首先检查是否有空闲连接，有就直接获取；其次检查连接数是否达上限，当达到上限后阻塞等待其它请求释放连接；最后当连接数未达到上限，主动创建连接。

步骤一如左图所示，从freeConn切片中获取连接，并检查连接是否过期

```
// Prefer a free connection, if possible.
numFree := len(db.freeConn)
if strategy == cachedOrNewConn && numFree > 0 {
    conn := db.freeConn[0]
    copy(db.freeConn, db.freeConn[1:])
    db.freeConn = db.freeConn[:numFree-1]
    conn.inUse = true
    db.mu.Unlock()
    if conn.expired(lifetime) {
        conn.Close()
        return nil, driver.ErrBadConn
    }
    conn.Lock()
    err := conn.lastErr
    conn.Unlock()
    if err == driver.ErrBadConn {
        conn.Close()
        return nil, driver.ErrBadConn
    }
    return conn, nil
}
```



获取连接

步骤二，创建channel，并加入到等待队列中，当其它请求释放连接时，将连接写入channel，通知阻塞请求。

```
if db.maxOpen > 0 && db.numOpen >= db.maxOpen {  
    // Make the connRequest channel. It's buffered so that the  
    // connectionOpener doesn't block while waiting for the req to be read.  
    req := make(chan connRequest, 1)  
    reqKey := db.nextRequestKeyLocked()  
    db.connRequests[reqKey] = req  
    db.waitCount++  
    db.mu.Unlock()  
  
    waitStart := time.Now()  
  
    // Timeout the connection request with the context.  
    select {  
    case ret, ok := <-req:  
        atomic.AddInt64(&db.waitDuration, int64(time.Since(waitStart)))  
  
        if !ok {  
            return nil, errDBClosed  
        }  
        if ret.err == nil && ret.conn.expired(lifetime) {  
            ret.conn.Close()  
            return nil, driver.ErrBadConn  
        }  
        if ret.conn == nil {  
            return nil, ret.err  
        }  
        // Lock around reading lastErr to ensure the session resetter finished.  
        ret.conn.Lock()  
        err := ret.conn.lastErr  
        ret.conn.Unlock()
```



获取连接

步骤三，调用注册驱动的

connect方法创建新连接，在连接创建失败后除将numOpen减1

外，然后调用

maybeOpenNewConnections方法

根据阻塞的请求数向openerCh

channel写入需创建的连接数。

调用此方法的原因是避免极端

情况下请求一直阻塞。

假设maxOpen值为1，请求A获取连接将numOpen值加1，请求B获取连接并阻塞在第二步，请求A创建新连接时失败，这时如果直接返回，请求B会一直阻塞。

```
db.numOpen++ // optimistically
db.mu.Unlock()
ci, err := db.connector.Connect(ctx)
if err != nil {
    db.mu.Lock()
    db.numOpen--
    db.maybeOpenNewConnections()
    db.mu.Unlock()
    return nil, err
}
db.mu.Lock()
dc := &driverConn{
    db:      db,
    createdAt: nowFunc(),
    ci:      ci,
    inUse:    true,
}
db.addDepLocked(dc, dc)
db.mu.Unlock()
return dc, nil

func (db *DB) maybeOpenNewConnections() {
    numRequests := len(db.connRequests)
    if db.maxOpen > 0 {
        numCanOpen := db.maxOpen - db.numOpen
        if numRequests > numCanOpen {
            numRequests = numCanOpen
        }
    }
    for numRequests > 0 {
        db.numOpen++ // optimistically
        numRequests--
        if db.closed {
            return
        }
        db.openerCh <- struct{}{}
    }
}
```




创建连接

在调用Open方法打开数据库时启动了一个用于创建新连接的协程，它阻塞在openerCh channel上，接收创建新连接事件，创建连接操作由openNewConnection方法完成。当有请求阻塞等待新连接时，无论新连接是否创建成功，都会唤醒一个阻塞的请求。

```
func (db *DB) openNewConnection(ctx context.Context) {
    ci, err := db.connector.Connect(ctx)
    db.mu.Lock()
    defer db.mu.Unlock()
    if db.closed {
        if err == nil {
            ci.Close()
        }
        db.numOpen--
        return
    }
    if err != nil {
        db.numOpen--
        db.putConnDBLocked(nil, err)
        db.maybeOpenNewConnections()
        return
    }
    dc := &driverConn{
        db:      db,
        createdAt: nowFunc(),
        ci:      ci,
    }
    if db.putConnDBLocked(dc, err) {
        db.addDepLocked(dc, dc)
    } else {
        db.numOpen--
        ci.Close()
    }
}
```



创建连接

```
func (db *DB) putConnDBLocked(dc *driverConn, err error) bool {  
    if db.closed {  
        return false  
    }  
    if db.maxOpen > 0 && db.numOpen > db.maxOpen {  
        return false  
    }  
    if c := len(db.connRequests); c > 0 {  
        var req chan connRequest  
        var reqKey uint64  
        for reqKey, req = range db.connRequests {  
            break  
        }  
        delete(db.connRequests, reqKey) // Remove from pending requests.  
        if err == nil {  
            dc.inUse = true  
        }  
        req <- connRequest{  
            conn: dc,  
            err:  err,  
        }  
        return true  
    } else if err == nil && !db.closed {  
        if db.maxIdleConnsLocked() > len(db.freeConn) {  
            db.freeConn = append(db.freeConn, dc)  
            db.startCleanerLocked()  
            return true  
        }  
        db.maxIdleClosed++  
    }  
    return false  
}
```



连接释放

连接在用完之后需归还给连接池以

实现连接复用，连接释放操作由
releaseConn方法来完成。

连接归还时首先检查连接在使用过程中是否出现错误，出错的连接不会被放入连接池中。然后调用
putConnDBLocked解除阻塞请求或
放入连接池。

```
func (dc *driverConn) releaseConn(err error) {
    dc.db.putConn(dc, err, true)
}

func (db *DB) putConn(dc *driverConn, err error, resetSession bool) {
    db.mu.Lock()
    if err == driver.ErrBadConn {
        db.maybeOpenNewConnections()
        db.mu.Unlock()
        dc.Close()
        return
    }
    if putConnHook != nil {
        putConnHook(db, dc)
    }

    added := db.putConnDBLocked(dc, nil)
    db.mu.Unlock()
}
```



连接过期

空闲连接超过

SetConnMaxLifetime方法
设置的最大生命周期后被
自动会释放，包使用惰性
释放策略，即每个生命周
期内只检查一次；除自动
检测外，每次调用
query/exec方法时都会检
查连接是否过期。

```
941 func (db *DB) connectionCleaner(d time.Duration) {  
942     const minInterval = time.Second  
943  
944     if d < minInterval {  
945         d = minInterval  
946     }  
947     t := time.NewTimer(d)  
948  
949     for {  
950         select {  
951             case <-t.C:  
952                 case <-db.cleanerCh:  
953             }  
954  
955             db.mu.Lock()  
956             d = db.maxLifetime  
957             if db.closed || db.numOpen == 0 || d <= 0 {  
958                 db.cleanerCh = nil  
959                 db.mu.Unlock()  
960                 return  
961             }
```



连接过期

```
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988

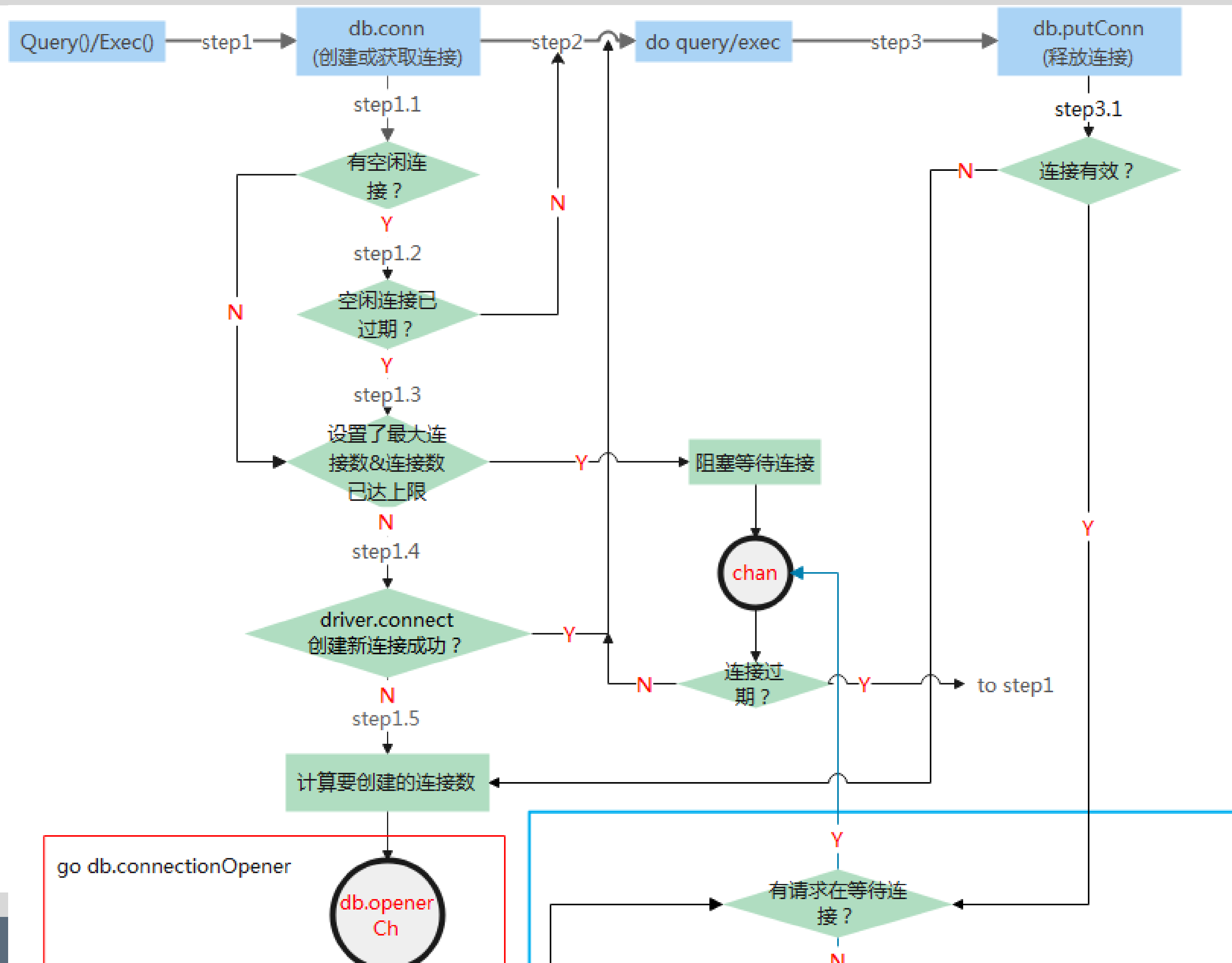
expiredSince := nowFunc().Add(-d)
var closing []*driverConn
for i := 0; i < len(db.freeConn); i++ {
    c := db.freeConn[i]
    if c.createdAt.Before(expiredSince) {
        closing = append(closing, c)
        last := len(db.freeConn) - 1
        db.freeConn[i] = db.freeConn[last]
        db.freeConn[last] = nil
        db.freeConn = db.freeConn[:last]
        i--
    }
}
db.maxLifetimeClosed += int64(len(closing))
db.mu.Unlock()

for _, c := range closing {
    c.Close()
}

if d < minInterval {
    d = minInterval
}
t.Reset(d)
}
```



database/sql组件连接池原理图。





05 | 名字服务

名字服务主要解决微服务架构中服务发现、配置共享等问题，它是服务治理的基石。名字服务一般有调用方、服务提供方和服务中心三种角色。

- 调用方主要功能：

服务发现

- 服务提供方主要功能：

服务注册

服务注销

服务保活

- 服务中心主要功能：

满足CP/AP模式

健康检查



服务中心

选用etcd来做为服务中心，它是一个高可用、强一致的 key-value 存储系统，属于CP模式，它具有简单、安全、写入速度快等特点，并提供watch、lease等机制可方便的实现服务发现和健康检查等功能。



服务注册

在向etcd注册服务前申请一个租约信息，对注册的服务设置TTL。其目的是避免服务异常结束后，调用方不觉能获取到其服务信息。

```
func (e *EtcdBuilder) registerLease(ctx context.Context, in *Instance) (err error) {  
    key := e.key(in.Name, in.Idc, in.PubEnv)  
    val, _ := json.Marshal(in)  
  
    // 申请Lease  
    ttlResp, err := e.client.Grant(context.TODO(), e.conf.LeaseTTL)  
    if err != nil {  
        return errors.Wrap(err, fmt.Sprintf("client.Grant failed, ins:%+v", in))  
    }  
  
    // 注册服务  
    _, err = e.client.Put(ctx, key, string(val), clientv3.WithLease(ttlResp.ID))  
    if err != nil {  
        return errors.Wrap(err, fmt.Sprintf("client.Put failed, key:%s, in:%+v", key, in))  
    }  
  
    return nil  
}
```



服务保活

服务注册时向etcd注册了一个带TTL的key，如果不采取其它手段，注册的信息很快就会过期。服务保活有两种方式，一种是续约，另一种是在过期前重新注册。

```
go func() {
    ticker := time.NewTicker(time.Duration(e.conf.LeaseTTL/3) * time.Second)
    defer ticker.Stop()
    for {
        select {
        case <-ticker.C:
            err := e.registerLease(ctx, in)
            if err != nil {
                zap.L().Error(fmt.Sprintf("%+v", err))
            }
        case <-ctx.Done():
            _ = e.unregister(in)
            ch <- struct{}{}
            return
        }
    }
}()
```



服务注销

注销比较简单，直接从etcd中删除key就行。

```
func (e *EtcdBuilder) unregister(ins *Instance) (err error) {
    key := e.key(ins.Name, ins.Idc, ins.PubEnv)
    if _, err = e.client.Delete(context.TODO(), key); err != nil {
        zap.L().Error(fmt.Sprintf("client.Delete failed, err:%+v", err),
            zap.String("key", key),
            zap.Any("ins", ins))
        return
    }

    zap.L().Info("client.Delete success", zap.String("key", key), zap.Any("ins", ins))
    return
}
```



服务发现

指通过服务名查找有效服务列表，并在服务有更新时主动通知调用方。etcd的watch机制可以发现服务的变动，并在变化时主动通知调用方。

```
func (srv *serverInfo) watch() {  
    _ = srv.getstore("get")  
    watchChan := srv.e.client.Watch(srv.e.ctx, srv.e.key(srv.sn), clientv3.WithPrefix())  
    for wresp := range watchChan {  
        for _, ev := range wresp.Events {  
            if ev.Type == mvccpb.PUT || ev.Type == mvccpb.DELETE {  
                _ = srv.getstore("watch")  
            }  
        }  
    }  
}
```



06 经验篇



06 | go.sum加入版本管理

go.sum包含依赖模块的校验和，构建项目时首先从cache(\$GOPATH/pkg/mod/cache)中找缺少的模块，如何命中则检查与go.sum中版本是否一致，如果未命中就下载并将校验和记录到go.sum文件中。

将go.sum提交到版本系统中，可以保证其它用户在构建项目时，使用相同版本的依赖。并每次向版本控制系统提交代码前，需执行go mod tidy命令更新项目依赖，然后再将go.sum提交到版本控制系统中。



06 | 类型别名

类型别名是golang 1.9版本的新特性，它的语法为：type identifier = Type, 与类型定义想非常相似，仅在identifier和Type之间多了一个=等号。但它俩有着本质的区别，由于golang是强类型语言，类型定义属于两种不同类型，不能直接赋值；类型别名两者之间共享方法和字段，两者属于同一类型。

类型别名常用来解决代码解耦，将一个大包分解为多个小包。

```
type MyInt1 int // 类型定义
type MyInt2 = int // 类型别名
```

```
var i int = 0
var i1 MyInt1 = i //error
var i2 MyInt2 = i // correct
```



06 | 类型别名

信用卡还款起初支持支付宝支付一种渠道，还款服务的代码如右上图所示，业务调用方代码如右下图所示。

```
type RepaymentService struct {}

func (this *RepaymentService) Baofu(orderId string, amount int) bool {
    // more...
}

func CreditRepayment(orderId string, amount int) {
    var r RepaymentService
    ret := r.Baofu(orderId, amount)
    // more...
}
```




06 | 类型别名

增加京东支付渠道的接入，由于京东支付与宝付是不同的渠道，因将两者的代码放在不同的文件或包中，在不改变已有业务调用方代码的前提下可使用类型别名来实现，如右上图所示。

业务调用方可继续调用

RepaymentService来完成还款，如右下图所示：

jdpay.go:

```
type JdPay = RepaymentService

func (this *JdPay) Jd(orderId string, amount int) bool {
    // more...
}
```

```
func CreditRepayment(orderId string, amount int) {
    var r RepaymentService
    ret := r.Jd(orderId, amount)
    // more...
}
```



06 | 测试http代码

在测试http代码时，可使用httptest包来代替创建http server，节省编码时间。如需要测试右图中的HealthCheckHandler接口是否正常工作。

```
func HealthCheckHandler(c *gin.Context) {  
    // more...  
    c.String(http.StatusOK, "alive")  
}  
  
func NewServer() *gin.Engine {  
    r := gin.Default()  
    r.GET("/health-check", HealthCheckHandler)  
  
    return r  
}  
  
func main() {  
    NewServer().Run()  
}
```



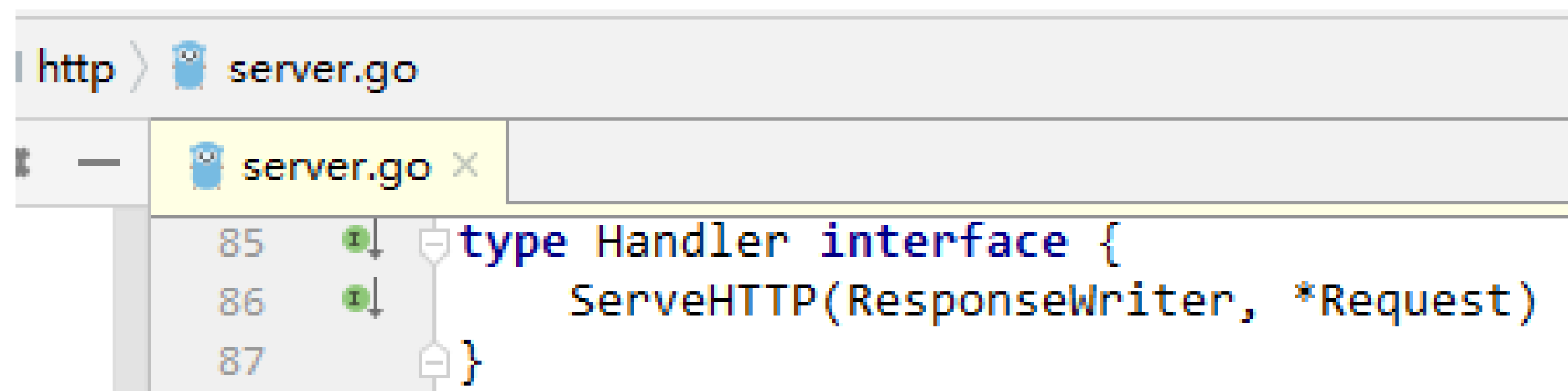
测试代码可以这么写，如右图所示：

```
func TestHealthCheckHandler(t *testing.T) {  
    // 使用httpptest创建http server  
    ts := httpptest.NewServer(NewServer())  
    defer ts.Close()  
  
    resp, err := http.Get(fmt.Sprintf("%s/health-check", ts.URL))  
    require.NoError(t, err)  
    defer resp.Body.Close()  
  
    require.Equal(t, resp.StatusCode, http.StatusOK)  
    bytes, err := ioutil.ReadAll(resp.Body)  
    require.NoError(t, err)  
    require.Equal(t, string(bytes), "alive")  
}
```



06 | 检测接口实现

在实现http服务时，可通过实现http.Handler接口来自定义请求处理逻辑。可在代码编译时检测接口实现是否正确来提前发现问题。



```
type MyServer struct {}

func (this *MyServer) ServeHTTP(http.ResponseWriter, *http.Request) {
    // more ...
}

// 检测Myserver是否正确实现http.Handler接口
var _ http.Handler = &MyServer{}
```



06 | 检测接口实现

GoLand IDE中查看接口

由那些类型实现：右击

接口名->Go To->

Implementation(s)

```
type Handler interface {
```

```
    Serve
```

Choose Implementation of **Handler** (81 found)

```
}
```

```
// A Resp
```

```
// constr
```

```
//
```

```
// A Resp
```

```
// has re
```

```
type Resp
```

```
    // He
```

```
    // Wr
```

```
    // Ha
```

```
    //
```

```
    // Ch
```

```
    // Wr
```

```
    // tr
```

```
    //
```

Counter in net/http/triv.go

Engine in gintest_main/vendor/github.com/gin-gonic/gin/gin.go

Engine in github.com/gin-gonic/gin/gin.go

Handler in cmd/go/internal/sumweb/server.go

Handler in github.com/micro/go-micro/api/handler/handler.go

Handler in github.com/micro/go-micro/api/handler/udp/udp.go

Handler in github.com/micro/go-micro/api/handler/unix/unix.go

Handler in golang.org/x/net/webdav/webdav.go

Handler in golang.org/x/net/websocket/server.go

Handler in net/http/cgi/host.go

HandlerFunc in net/http/server.go

MyServer in modtest-main/main.go

PauseableHandler in go.etcd.io/etcd/pkg/testutil/pauseable_handler.go



php中有丰富的函数库，在GO中要如何实现？

https://www.php2golang.com

☆

PHP » GoLang

GoLang alternatives for PHP functions

Enter some PHP function, class or library name...

substr

Search

Recently updated

[more](#)

[function.strpos](#) (Dec/29)

[function.intval](#) (Dec/17)

[function.mb-strpos](#) (Dec/16)

Most requested

[more](#)

[dateinterval.createfromdatestring](#) (1464 times)

[function.sort](#) (370 times)

[function.stream-notification-callback](#) (287 times)

Last requests

[more](#)

[splfileobject.current](#) (Jan/11)

[gmagick.readimagefile](#) (Jan/11)

[function.newt-checkbox-set-flags](#) (Jan/11)



封装可以隐藏实现细节，提高代码的可用性和可维护性。golang中实现封装的方式与其它OOP语言不一样，golang中的每个包都有一个包名，包名的作用相当于名字空间，包通过控制包内成员是否导出来实现封装特性。

Go中大写字母开头的标识符定义为包外可访问，小写的则不会。这种限制同样适用于struct和类型的方法。

当一个包的实现太复杂时，我们考虑将其分成多个小包，但小包里的实现细节除父包外，又不想暴露给其它外部包。Go提供了内部包机制来帮助我们实现此功能。



06 | 封装

当包的路径中包含internal名字时，这种包被称为“内部包”，go的构建工具会对其做特殊处理。一个内部包只能被有同一个父目录的包导入。如：

net/http/internal包可以被

net/http/httptest包和net/http包导入，但不能被net/rpc包导入。

The screenshot shows a Go IDE with a file explorer on the left and a code editor on the right. The file explorer displays a directory structure for the 'net/http' package, including subdirectories like 'cgi', 'cookiejar', 'fcgi', and 'httptest'. The 'httptest' directory is expanded, showing files like 'example_test.go', 'httptest.go', 'httptest_test.go', 'recorder.go', 'recorder_test.go', 'server.go' (which is selected), and 'server_test.go'. Below 'httptest' are 'httptrace', 'httputil', and 'internal' directories. The 'internal' directory is expanded, showing 'chunked.go', 'chunked_test.go', and 'testcert.go'. The code editor on the right shows the source code of 'server.go'. It starts with a copyright notice, followed by a package declaration 'package httptest', an import block for various packages including 'crypto/tls', 'crypto/x509', 'flag', 'fmt', 'log', 'net', 'net/http', 'net/http/internal', 'os', 'strings', 'sync', and 'time', and then a type definition 'type Server struct {'.

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Implementation of Server.
6
7 package httptest
8
9 import (
10     "crypto/tls"
11     "crypto/x509"
12     "flag"
13     "fmt"
14     "log"
15     "net"
16     "net/http"
17     "net/http/internal"
18     "os"
19     "strings"
20     "sync"
21     "time"
22 )
23
24 // A Server is an HTTP server.
25 // Local loopback interface.
26 type Server struct {
```




06 | 继承

Go语言中没有类，也没有类层次结构的概念，它通过组合来实现继承特性。组合是Go语言中面向对象编程的核心思想，它通过struct中的匿名字段来实现。组合可以嵌套，并支持方法覆盖。

```
type Foo struct{}

func (this *Foo) String() string {
    return "foo"
}

type Bar struct{}

func (this *Bar) String() string {
    return "bar"
}

type FooBar struct {
    Foo
    Bar
}

func (this *FooBar) String() string {
    return "foobar"
}

func main() {
    var o FooBar
    fmt.Println(o.String())
    fmt.Println(o.Foo.String())
    fmt.Println(o.Bar.String())
}
```



golang中，一个struct实现了某个接口中的所有方法，那么我们就认为，该struct实现了此接口。我们可以将实现了接口的类型变量赋值给接口变量，以此来实现运行时多态。

```
io > io.go
ocodes
88 //
89 // Implementations must not retain p.
90 type Writer interface {
91     Write(p []byte) (n int, err error)
92 }
...
type MyWrite struct {}

func (w MyWrite) Write(p []byte) (n int, err error) {
    fmt.Println(string(p))
    return len(p), nil
}

func Write(o io.Writer, data string) {
    w := bufio.NewWriter(o)
    w.WriteString(data)
    w.Flush()
}

func main() {
    f, _ := os.Create("output.txt")
    defer f.Close()
    Write(f, "write to file")

    var my MyWrite
    Write(my, "write to stdout")
}
```



06 | goroutine泄漏

导致goroutine泄漏的主要原因死循环和阻塞，如读一个未写过的unbuffered channel、向已满的buffered channel中写等等。当发现goroutine泄漏时我们可通过pprof查出异常的goroutine。

图中实例发起了10次http调用，请求失败后将err写入unbuffered channel中，channel没有读取方，http请求的goroutine都会被阻塞。

```
import (  
    "fmt"  
    "net/http"  
    "net/http/pprof"  
    "time"  
)  
  
func Handler(w http.ResponseWriter, r *http.Request) {  
    w.WriteHeader(500)  
    w.Write([]byte("failed"))  
}  
  
func main() {  
    ch := make(chan error)  
    http.HandleFunc("/", Handler)  
  
    go func() {  
        time.Sleep(time.Second)  
        for i := 0; i < 10; i++ {  
            go func(url string) {  
                resp, err := http.Get(url)  
                if err != nil {  
                    ch <- err  
                    return  
                }  
                defer resp.Body.Close()  
            }(fmt.Sprintf("/test%d", i))  
        }  
    }()  
}
```



06 | goroutine泄漏

我们使用pprof看看goroutine都阻塞在那。

```
< > ↻ ⓘ 不安全 | 192.168.126.10:8000/debug/pprof/goroutine?debug=1

goroutine profile: total 14
10 @ 0x432590 0x40693b 0x406911 0x4066f5 0x753c78 0x45f741
#      0x753c77      main.main.func1.1+0x107 /codes/golang/src/modtest-main/main.go:27

1 @ 0x432590 0x42d32a 0x42c8f5 0x4c0325 0x4c129f 0x4c1281 0x59483f 0x5a7ea8 0x6bd284 0x54a963 (
#      0x42c8f4      internal/poll.runtime_pollWait+0x54      /usr/local/go1.13.6/src
#      0x4c0324      internal/poll.(*pollDesc).wait+0x44      /usr/local/go1.13.6/src
#      0x4c129e      internal/poll.(*pollDesc).waitRead+0x1ce /usr/local/go1.13.6/src
#      0x4c1280      internal/poll.(*FD).Read+0x1b0           /usr/local/go1.13.6/src
#      0x59483e      net.(*netFD).Read+0x4e                   /usr/local/go1.13.6/src
#      0x5a7ea7      net.(*conn).Read+0x67                    /usr/local/go1.13.6/src
#      0x6bd283      net/http.(*connReader).Read+0xf3         /usr/local/go1.13.6/src
#      0x54a962      bufio.(*Reader).fill+0x102               /usr/local/go1.13.6/src
#      0x54b6bc      bufio.(*Reader).ReadSlice+0x3c           /usr/local/go1.13.6/src
#      0x54b8f3      bufio.(*Reader).ReadLine+0x33            /usr/local/go1.13.6/src
#      0x64450b      net/textproto.(*Reader).readLineSlice+0x6b /usr/local/go1.13.6/src
#      0x6b7861      net/textproto.(*Reader).ReadLine+0x91    /usr/local/go1.13.6/src
#      0x6b7890      net/http.readRequest+0xc0                /usr/local/go1.13.6/src
#      0x6be57e      net/http.(*conn).readRequest+0x15e       /usr/local/go1.13.6/src
#      0x6c2b03      net/http.(*conn).serve+0x6d3             /usr/local/go1.13.6/src
```

```
< > ↻ ⓘ 不安全 | 192.168.126.10:8000/debug/pprof/goroutine?debug=2

goroutine 34 [chan send]:
main.main.func1.1(0xc000078120, 0xc00010e018, 0x6)
    /codes/golang/src/modtest-main/main.go:27 +0x108
created by main.main.func1
    /codes/golang/src/modtest-main/main.go:24 +0xcc

goroutine 35 [chan send]:
main.main.func1.1(0xc000078120, 0xc00010e028, 0x6)
    /codes/golang/src/modtest-main/main.go:27 +0x108
created by main.main.func1
    /codes/golang/src/modtest-main/main.go:24 +0xcc

goroutine 36 [chan send]:
main.main.func1.1(0xc000078120, 0xc00010e038, 0x6)
    /codes/golang/src/modtest-main/main.go:27 +0x108
created by main.main.func1
    /codes/golang/src/modtest-main/main.go:24 +0xcc

goroutine 37 [chan send]:
main.main.func1.1(0xc000078120, 0xc00010e048, 0x6)
    /codes/golang/src/modtest-main/main.go:27 +0x108
created by main.main.func1
    /codes/golang/src/modtest-main/main.go:24 +0xcc
```



Thanks