一、数组类型-garray

`garray`支持`int`/`string`/`interface{}`三种常用的数据类型

1. garray

   支持普通数组和排序数组，普通数组的结构体名称定义为

   ```
   *Array
   ```

   格式，排序数组的结构体名称定义为

   ```
   Sorted*Array
   ```

   格式，如下：

   - `Array`，`intArray`，`StrArray`
   - `SortedArray`，`SortedIntArray`，`SortedStrArray`
   - 其中排序数组`SortedArray`，需要给定排序比较方法，在工具包`gutil`中也定义了很多`Comparator*`比较方法

```go
package main

import (
    "fmt"
    "github.com/gogf/gf/v2/container/garray"
)


func main () {
    // 创建并发安全的int类型数组
    a := garray.NewIntArray(true)

    // 添加数据项
    for i := 0; i < 10; i++ {
        a.Append(i)
    }

    // 获取当前数组长度
    fmt.Println(a.Len())

    // 获取当前数据项列表
    fmt.Println(a.Slice())

    // 获取指定索引项
    fmt.Println(a.Get(6))

    // 在指定索引后插入数据项
    a.InsertAfter(9, 11)
    // 在指定索引前插入数据项
    a.InsertBefore(10, 10)
```

```
    fmt.Println(a.Slice())

    // 修改指定索引的数据项
    a.Set(0, 100)
    fmt.Println(a.Slice())

    // 搜索数据项，返回搜索到的索引位置
    fmt.Println(a.Search(5))

    // 删除指定索引的数据项
    a.Remove(0)
    fmt.Println(a.Slice())

    // 并发安全，写锁操作
    a.LockFunc(func(array []int) {
        // 将末尾项改为100
        array[len(array) - 1] = 100
    })

    // 并发安全，读锁操作
    a.RLockFunc(func(array []int) {
        fmt.Println(array[len(array) - 1])
    })

    // 清空数组
    fmt.Println(a.Slice())
    a.Clear()
    fmt.Println(a.Slice())
}
```

输出：

```
10
[0 1 2 3 4 5 6 7 8 9]
6 true
[0 1 2 3 4 5 6 7 8 9 10 11]
[100 1 2 3 4 5 6 7 8 9 10 11]
5
[1 2 3 4 5 6 7 8 9 10 11]
100
[1 2 3 4 5 6 7 8 9 10 100]
[]
```

二、完全理解Logic方法包的初始化

```
type sBookTypeQuery struct{}

func init() {
   service.RegisterBookTypeQuery(New())
}
// RegisterBookTypeQuery注册的是谁？这里：
//func RegisterUser(i IUser) {
//  localUser = i
//}

func (s *sBookTypeQuery) BookTypeQuery(ctx context.Context) () {
   return
}
```

1. 定义 sBookTypeQuery 结构体：

```
type sBookTypeQuery struct{}
```

定义一个空的结构体 `sBookTypeQuery`。这个结构体是实现了 `service.BookTypeQuery` 接口的一部分。

结构体的主要作用是**为后续的方法提供一个接收者**，(即将结构体实例作为方法的接收者 --- 绑定)以便我们可以在方法内部操作结构体的数据和状态。它可以包含字段和方法，用于处理业务逻辑和实现特定的功能。

优势：

1. **封装和组织：** 通过将方法与结构体关联，可以将操作数据的逻辑与数据本身封装在一起，提高代码的组织性和可维护性。
2. **数据共享：** 方法内部可以访问结构体的私有字段，从而实现对数据的共享和操作，而无需暴露数据的实现细节。
3. **代码复用：** 可以在多个方法中重复使用相同的操作逻辑，提高代码复用率。
4. **扩展性：** 可以轻松地添加新的方法来扩展结构体的功能，而不需要改变已有的代码。

2. 初始化函数 init：

```
func init() {
    service.RegisterBookTypeQuery(New())
}
```

`init()` 函数是在包被引入时自动执行的函数。在这里，它的主要作用是在应用程序初始化过程中注册服务。具体地，它调用了 `service.RegisterBookTypeQuery` 函数，并将 `New()` 返回的对象作为参数传递给它。这样可以在其他地方通过已注册的服务接口调用相关功能，实现代码解耦和模块化。

3. New 函数：

```
func New() *sBookTypeQuery {
    return &sBookTypeQuery{}
}
```

`New()` 函数是一个工厂函数，用于创建并返回一个 `sBookTypeQuery` 结构体的实例，简单地返回一个新创建的 `sBookTypeQuery` 结构体指针。

> **个人总结：**

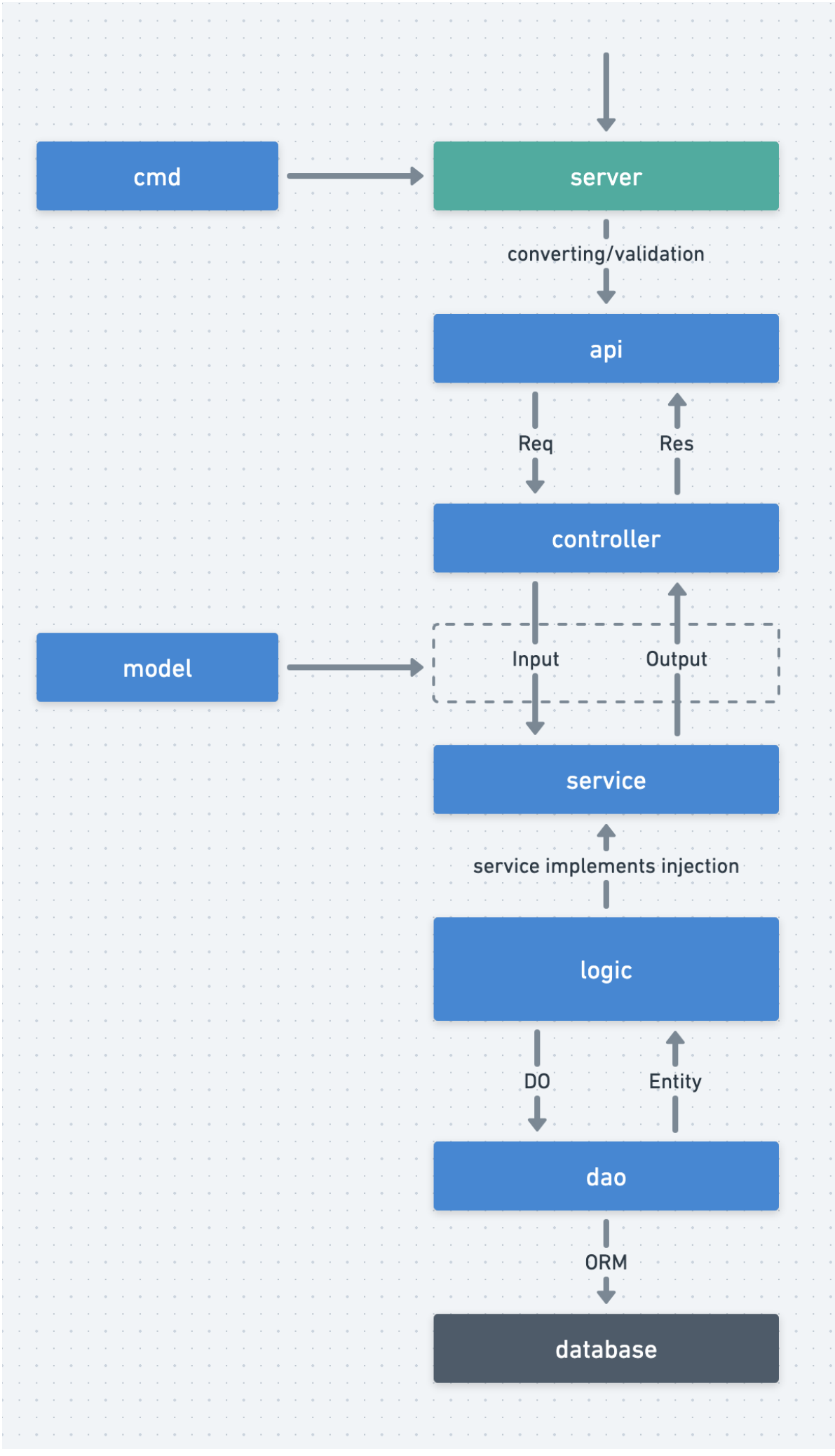
首先创建一个结构体，该结构体是空的，说明它不是用来存储数据的，而是用来关联数据的。我们常看到，多个方法调用同一个结构体，并在内部对其进行“操作”，但是反过来一想，在这种情况下，一个结构体也与多个方法产生了关联（绑定），毕竟力是相互的嘛！你懂的~。

然后是一个初始化init函数，主要作用是在应用程序初始化过程中注册服务。为啥要注册这么一下子？因为它调用了 `service.RegisterBookTypeQuery` 函数，并将 `New()` 返回的对象作为参数传递给它。这样可以在其他地方通过已注册的服务接口调用相关功能，实现代码解耦和模块化。

最后是一个New函数，而且是指针类型的。主要作用1是为上面创建的结构体实例化一下子，2是New 函数应该返回实例的指针，以便在方法调用中能够修改实例的状态。

三、服服子

> 每当添加新的logic文件后，要在/internal/logic下的logic文件里引用


cmd
server
converting/validation
api
Req
Res
controller
model
Input
Output
service
service implements injection
logic
DO
Entity
dao
ORM
database

四、彻底搞懂数据库DAO

DAO（Data Access Object）是一种设计模式，用于将数据访问逻辑与业务逻辑分离。DAO层通常负责处理数据的持久化操作，即与数据库进行交互，执行数据的增删改查等操作。相当于一个壳子，把数据库包起来了，你想访问它，就得通过DAO。

五、gf gen service

> **为什么想重新生成注册信息的时候，要把new()去除和加上后各执行一遍?**

```
func New() *sUserInformation {
   return &sUserInformation{}
}
```

gf gen service是在 `生成接口及服务注册文件`，之后在每个业务模块中加上接口的具体实现注入。该方法每个业务模块加一次即可。也就是：

```
func init() {
    service.RegisterBookTypeQuery(New())
}
```

因此，需要用gf gen service重新生成一下

六、搞懂不同权限的功能分配

问题：普通用户登录和管理员用户登录后的功能是不同的，用户的功能和管理员的功能会有重合的，那么他们是直接复用一个功能接口，还是有各自的接口?

如果直接复用一个功能接口，当用户调用该请求时，返回的时候返回什么?

重置WMI服务

```
winmgmt /resetrepository
```

七、搞懂GoFrame框架状态码

八、搞透GoFrame框架注册中间件

九、搞透数据库中的锁

当多个用户或线程 `同时` 访问数据库时，为了维护数据的 `一致性和完整性`，数据库系统需要实施锁机制。锁机制确保在**某个时刻**只有**一个用户或线程**可以**访问某个数据**，以防止并发操作引发的问题。

锁机制： MySQL的锁机制用于控制并发访问，它可以分为两类主要的锁：**共享锁**（也称为**读锁**）和**排他锁**（也称为**写锁**）。

1. **共享锁（S锁）**：多个用户可以同时获得共享锁，允许多个用户同时读取数据**（读读不冲突）**，但阻止其他用户获取排他锁，从而防止写操作（**读写冲突**）。适用于读多写少的情景。
2. **排他锁（X锁）**：只有一个用户可以获得排他锁，允许用户进行写操作，但阻止其他用户获得任何类型的锁（**写写冲突**），包括共享锁和排他锁。适用于写操作频繁的情景。

> 好玩的：
>
> **乐观锁**和**悲观锁**更多的是体现加锁的思想不同，乐观锁是一种无锁的思想，**假设并发冲突总是不会发生**，提交时检查数据一致性，如果一致性被破坏则放弃提交，更新时带着 version 就是乐观锁。**悲观锁假设并发冲突一定会发生**，每次操作前都会拿锁，通过锁的互斥顺序执行来控制并发，**可以认为数据库中的锁都是悲观锁。**

共享锁示例:

(当事务对数据加上读锁后，其他事务只能对该数据加读锁，不能加写锁。)

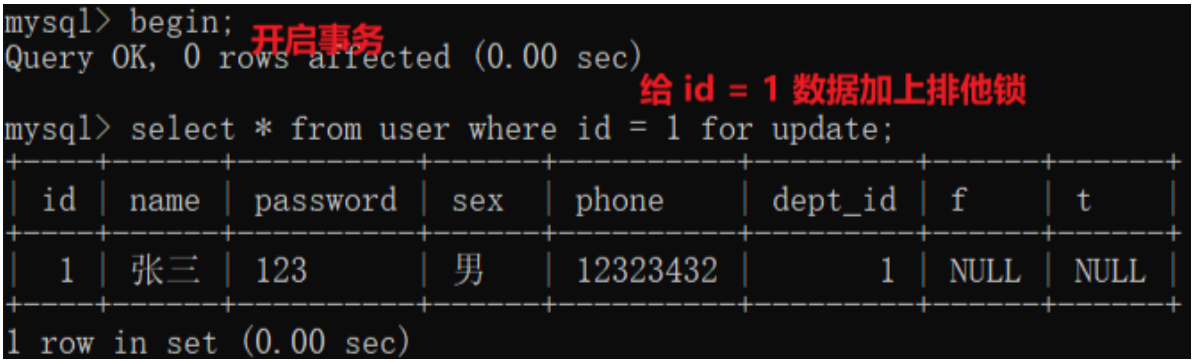
用户1：

用户2：

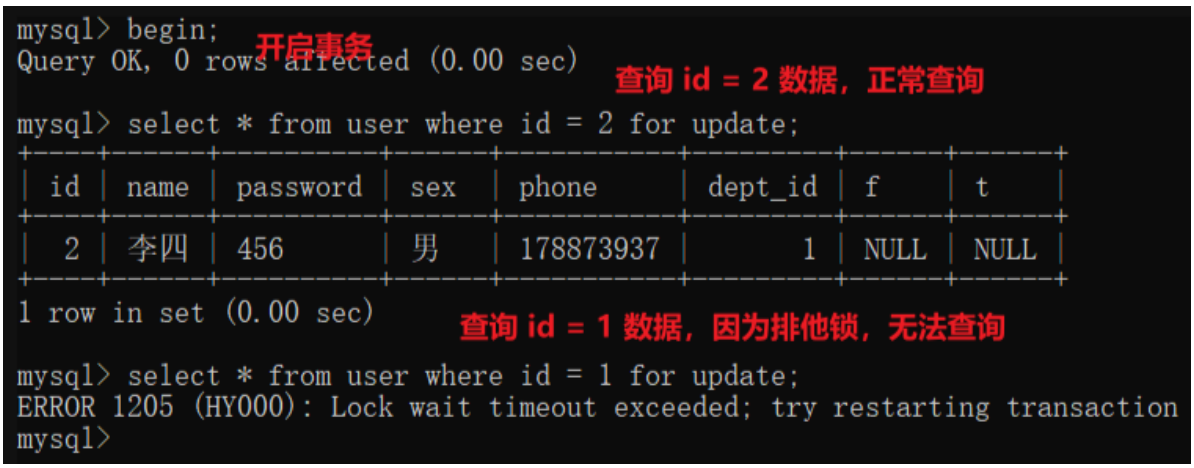
排他锁示例

当事务对数据加上排他锁后，其他事务无法对该数据进行查询或者修改（查都不让查。。。。）

用户1：



用户2：



引用：https://blog.csdn.net/wgzblog/article/details/127281354

为什么有了事务这东西，还需要乐观锁悲观锁?

事务是粗粒度的概念、乐观锁悲观锁可以更细粒度的控制；比如抢票，假设余票只有1张；隔离级别可以保证事务A和事务B不能读到对方的数据，也不能更新对方正在更新的数据，但是事务A和事务B都认为还有1张余票，于是出票，并更新为0；

十、`service`目录接口代码自动生成注意事项

logic目录下具体代码文件的结构体必须以小写字母s开头，例如sBorrowHistory，否则无法自动生成service接口。

对了，service接口是干啥的呢?

为了把业务逻辑进行封装，形成一个接口目录。

```
// 里面全是接口
type (
   IBookType interface {
      // 图书类别查询方法
      BookTypeQuery(ctx context.Context, req *v1.BookTypeReq) (res
*v1.BookTypeRes, err error)
      BookTypeNumQuery(ctx context.Context, BookTypeID int) (res
*v1.BookTypeNumRes, err error)
   }
)
```

用正则指定业务模块结构体定义格式，便于解析业务接口定义名称。在默认的正则下，所有小写`s`开头，大写字母随后的结构体都将被当做业务模块接口名称。例如：

logic结构体名称	service接口名称
`sUser`	`User`
`sMetaData`	`MetaData`