lab\_common\_so说明及使用文档

# 版本

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **修改内容** | **作者** |
| 2015/08/18 | V1.0 | 全文 |  |

# 1 简介

lab\_common\_so是一个C/C++语言编写的，基于woo框架（一个轻型通讯框架，通讯协议及日志系统比较完善）的高效计算服务。

lab\_common\_so具备如下特性:

* 透明化的数据获取
* 数据本地化
* 非中断式的资源实时更新
* 统一的接口形态
* 规范化的业务流程
* 插件式的业务更新
* 可配置的算法支持

lab\_common\_so的使用场景包括但不限于：

* 高效的实时在线计算服务
* 并发式的离线计算服务
* 数据存储代理
* 算法模型训练

# 2 框架结构

在框架中，每个线程包括一个work\_company对象的指针和一个db\_company对象的指针。其中work\_company用于管理业务，db\_company用户管理数据资源。

## 2.1 db\_company

包含一个global\_db\_company，同时还记录了db\_interface映射关系。

（1）global\_db\_company

记录了全局数据资源，所有线程共享。

（2）db\_interface映射关系

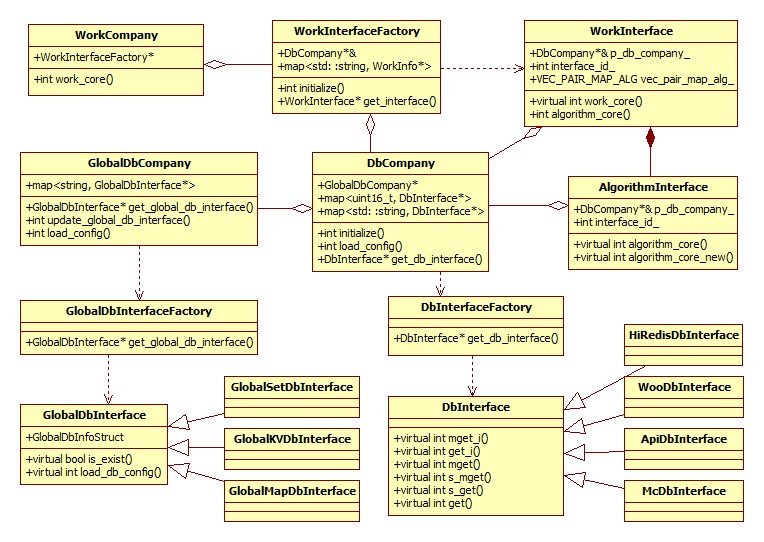
记录了db\_id以及db\_name和db\_interface的对应关系，使用时可根据配置文件中的DB\_ID字段或者DB\_NAME字段获取对应的db\_interface。

## 2.2 work\_company

包含一个work\_interface工厂类对象，以及一个入口函数work\_core。

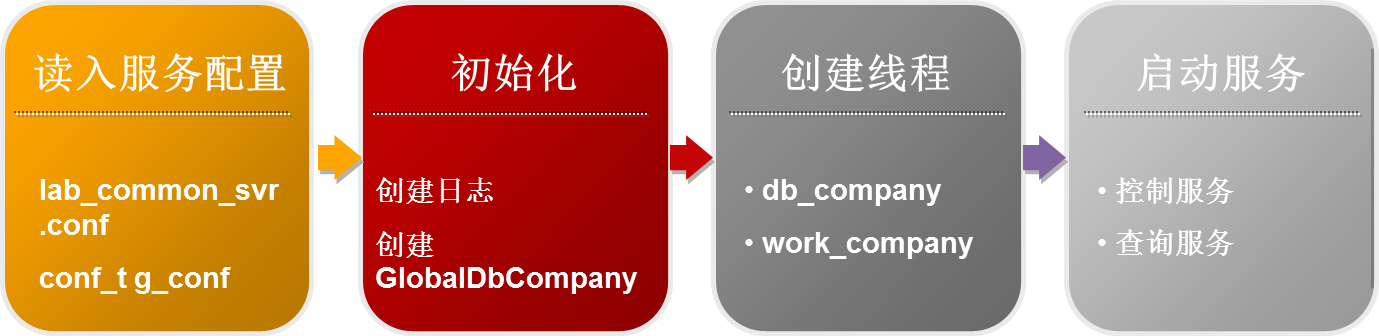
工厂对象包含了业务名称与业务的对应关系，根据业务名称，获取对应的业务，调用业务的work\_core函数执行业务。

## 2.3 框架类图



从类图中可以看到，框架的核心在于DbCompany和WorkCompany。而GlobalDbInterface、DbInterface、WorkInterface和AlgorithmInterface是本框架的基础。

## 2.4 服务启动流程图



# 3 框架详解

本节将从GlobalDb、Db、Work和Algorithm四种基础类型介绍框架的实现。并通过服务处理流程的介绍，说明各个类型之间的关联关系。

## 3.1 GlobalDb类型

该类型用于支持全局静态数据的载入及获取。数据多以文件形式挂在本地磁盘，在框架启动时加载入内存，全局唯一，所有线程共享。

GlobalDbInterface是所有全局静态数据读取类的基类，提供了一个is\_exist通用函数接口用于查询指定key是否在数据库中。该类的某些派生类还提供了get\_value和mget\_value等特定函数接口。

目前支持如下类型：

* \_\_gnu\_cxx::hash\_set<uint64\_t>
* \_\_gnu\_cxx::hash\_map<uint64\_t, uint32\_t>
* MapDb（自研）

## 3.2 Db类型

该类型用于支持远程数据的获取。

DbInterface是所有远程数据读取类的基类，目前派生出了四种类型：

* redis：提供对远程redis数据库的访问
* woo：提供对基于woo协议服务的访问
* openAPI：提供对http服务的访问
* MC：提供对远程memcache数据库的访问

目前提供了四个通用函数接口：

* get(uint64\_t n\_key)：根据整型key访问数据库
* s\_get(char\* p\_str\_key)：根据字符串key访问数据库
* mget(uint64\_t n\_keys[])：根据多个整型key访问数据库
* s\_mget(char\* p\_str\_keys[])：根据多个字符串key访问数据库

此外，还提供了一个get\_multi\_db函数，可对多种数据请求，并行访问不同数据库。

## 3.3 Work类型

该类型是业务类，提供给用户进行二次开发。

框架对业务执行了so化，使得业务可以快速上线迭代，并能让服务支持多个业务。

我们定义了统一的接口格式，在基类work\_interface中提供了统一的接口函数work\_core。

对于输入的请求串，必须是json格式的，例如{"api":"example", "cmd":"query", "body":"welcome!"}。其中api是必需的，以此来获取指定的业务，并将请求中的其它参数传递给业务。对于输出格式，没有进行限制。

## 3.4 Algorithm类型

该类型是算法类，提供给用户进行二次开发。

框架对算法执行了so化，使得算法库可以迅速上线，并能让多个业务使用多个算法库。

我们定义了统一的接口格式，在基类algorithm\_interface中提供了统一的接口函数algorithm\_core。

## 3.5 服务类型

框架可编译成两个可执行文件，分别为lab\_common\_main和lab\_common\_svr。

lab\_common\_main程序是离线处理程序，执行完请求之后会退出。

lab\_common\_svr程序是在线服务程序，持续监听端口，接收请求并处理。

目前在线程序提供两种服务，一种是查询服务，一种是控制服务。

查询服务：接收到业务请求，进行处理，返回处理后的结果。

控制服务：对服务进行控制，目前支持静态数据更新，远程数据资源配置更新，业务配置更新等。

## 3.6 查询服务处理流程

服务的请求处理流程如下：

第一步：对于一次请求，服务的某个线程使用work\_company工厂对象的work\_core函数；

第二步：对传入的请求参数进行解析，选择合适的业务处理类。

第三步：调用业务处理类的work\_core函数进行处理。

第四步：在业务处理类中，调用db\_interface获取远程数据，调用global\_db\_interface获取本地数据，调用algorithm\_core函数执行算法。

第五步：将执行完的结果，拼装成某种格式（一般使用json）,返回。

流程图如下：

work\_company解析请求

调用合适的业务处理类

获取远程数据

获取本地数据

调用算法

格式化结果

请求

结束

# 4 目录结构及文件说明

（注：如下说明中，斜体文本表示该文件是单个文件，非斜体部分表示该文件是目录）

* alg\_tool：算法开发目录
  + *Makefile*
  + *Makefile.in*
  + src 算法源码文件目录
* bin：可执行文件目录
  + *binserverclient.py* python版woo框架客户端程序
  + *memleak.sh* 内存泄露测试脚本
  + *restartwork.sh* 更新业务脚本
  + *run.sh* 服务启动脚本
  + *saferun* 监控脚本，当目标程序被终止，会自动重启目标程序
  + *test.sh* 服务测试脚本
* conf：配置文件目录
  + alg 算法配置文件目录，写法参考【附件6】
  + *db\_config.ini* 远程数据配置文件，写法参考【附件1】
  + *global\_db\_config.ini* 本地数据配置文件，写法参考【附件2】
  + *lab\_common\_main.conf* lab\_common离线运算配置文件，写法参考【附件5】
  + *lab\_common\_svr.conf* lab\_common服务配置文件，写法参考【附件4】
  + *work\_config.ini* 业务配置文件，写法参考【附件3】
  + *work\_config.ini.start* 业务配置文件，用于../bin/restartwork.sh
  + *work\_config.ini.stop* 业务配置文件，用于../bin/restartwork.sh
* data：本地数据文件存放目录
* depend：依赖库或者依赖外部程序
  + feature\_pool\_bin 自研的featurepooldb的依赖文件以及.a库
  + mapdb 自研的mapdb的依赖文件以及.a库
  + uthash 提供给mapdb使用的头文件
  + woo 自研的轻型通讯框架，通讯协议及日志系统比较完善
  + tools 外部依赖库，包括curl，redis，memcache，json等
* include：基础类及工具类目录
  + algorithm：算法基础类
    - *algorithm\_func.h* 算法so构造
    - *algorithm\_interface.h* 算法接口类
* db：数据基础类（为描述清晰，本节不按照文件名排序）
  + - *db\_company.h* 数据组织者
    - *global\_ db\_company.h* 本地数据组织者
    - *db\_interface\_factory.h* 远程数据工厂类
    - *db\_interface.h* 远程数据抽象基类
    - *api\_db\_interface.h* 远程数据派生类，提供对http的访问
    - *hiredis\_db\_interface.h* 远程数据派生类，提供对redis的访问
    - *mc\_db\_interface.h* 远程数据派生类，提供对memcache的访问
    - *woo\_db\_interface.h* 远程数据派生类，提供对woo协议服务的访问
    - *global\_db\_interface\_factory.h* 本地数据工厂类
    - *global\_db\_interface.h* 本地数据抽象基类
    - *global\_set\_db\_interface.h* 本地数据派生类，提供对集合的访问
    - *global\_kv\_db\_interface.h* 本地数据派生类，提供对键值对的访问
    - *global\_map\_db\_interface.h* 本地数据派生类，提供对mapdb的访问
* utility：工具类目录
  + *base\_define.h* 基础数据定义
  + *cand\_user.h* 为算法函数定义的结构
  + *encode\_convert.h* 编码转化头文件
  + *ini\_file.h* 配置文件解析头文件
  + *utility.h* 工具类
  + work：业务基础类
    - *work\_company.h* 业务组织者
    - *work\_interface\_factory.h* 业务工厂类
    - *work\_interface.h* 业务抽象基类
* lib：业务及算法编译后生成的so文件
* log：日志存放目录
* *Makefile*
* *Makefile.in*
* result：离线运算结果目录
* *setup\_dependent.sh*：安装依赖库脚本
* src：主程序目录
  + *binaryserver.cpp* woo框架客户端实现文件
  + *encode\_convert.cpp* 编码转化实现文件
  + *ini\_file.c* 配置文件解析实现文件
  + *lab\_common\_main.cpp* 离线运算实现文件
  + *lab\_common\_svr.cpp* 在线服务实现文件
* work\_tool：业务开发目录
  + *Makefile*
  + *Makefile.in*
  + src 业务源码文件目录

# 5 如何开始

（1）执行sh ./setup\_dependent.sh 安装依赖项

（2）cd ./alg\_tool; make 编译生成算法库

（3）cd ../work\_tool; make 编译生成业务库

（4）cd ..; make 编译生成主程序

（5）cd ./bin; sh run.sh 启动服务

（6）cd ./bin; sh test.sh 运行测试程序，观察结果为“welcome”

可能出现的问题

* json安装失败

请删除/usr/local/include/json以及json-c，单独安装json-c-master

* memcache安装失败

请单独安装

* curl安装失败

请单独安装

* 访问异常，请检查服务是否启动成功。

在log目录下，有两个日志文件。

lab\_common.log记录服务处理日志，出现“query server started”表明服务启动成功。

work\_interface.log记录服务进程，若服务正常启动，则该日志内容不会变动。

# 6 如何开发

## 6.1 新建一个业务

可以参考work\_tool/src目录的ExampleWorkInterface类或者SimpleFrameWorkInterface类来编写自己的业务类。

### 6.1.1 业务开发流程

第一步：继承或间接继承WorkInterface类

第二步：实现虚函数work\_core

第三步：work\_tool目录下，修改Makefile

第四步：编译生成so文件，可在lib目录下查看

第五步：修改conf/work\_config.ini，添加新增业务

第六步：启动服务，若服务已启动，也可采用更新业务的形式，见6.8节

### 6.1.2 访问远程数据

|  |
| --- |
| // 使用db\_company获取对应的db接口类  DbInterface\* p\_remote\_db\_interface = p\_db\_company\_->get\_db\_interface("EXAMPLE\_REDIS\_DB ");  // 使用db接口类，调用合适的方法获取数据  int result = p\_remote\_db\_interface->get(0, uid, p\_result, split\_char, second\_split\_char);  // TODO 数据结果存放在p\_result中，进行处理 |

注：该步骤可能需要配置conf/db\_config.ini

### 6.1.3 访问本地数据

|  |
| --- |
| // 使用db\_company获取对应的global\_db\_company  GlobalDbCompany\* p\_g\_db\_company = p\_db\_company\_->get\_global\_db\_company();  // 使用global\_db\_company获取对应的global\_db接口类  GlobalDbInterface\* g\_interface = p\_g\_db\_company->get\_global\_db\_interface("EXAMPLE\_SET\_DB ");  // 使用global\_db接口类，调用合适的方法获取数据  return g\_db\_interface->is\_exist(uid); |

注：该步骤可能需要配置conf/global\_db\_config.ini

### 6.1.4 调用算法

|  |
| --- |
| VEC\_CAND result\_vec; // 指定格式的候选集合  AccessStr access\_str; // 指定格式的访问结构  access\_str.uid\_ = uid;  // 调用算法库  int result = algorithm\_core(req\_id, access\_str, result\_vec); |

注：改步骤可能需要在conf/alg目录下新增业务配置，并在conf/work\_config.ini中添加算法配置路径

## 6.2 新建一个远程数据资源访问类

第一步：继承或间接继承DbInterface类

第二步：实现DbInterface类的纯虚函数

第三步：在db\_interface.h的枚举类型DB\_TYPE中添加新类型

第四步：在db\_interface\_factory.h中引入新类，并在函数get\_db\_interface中添加新case

## 6.3 新建一个本地数据资源访问类

第一步：继承或间接继承GlobalDbInterface类

第二步：实现GlobalDbInterface类的纯虚函数is\_exist、load\_db\_config

第三步：在global\_db\_interface.h的枚举类型GLOBAL\_DB\_TYPE中添加新类型

第四步：在global\_db\_interface\_factory.h中引入新类，并在函数get\_global\_db\_interface中添加新case

## 6.4 新建一个算法

第一步：继承或间接继承AlgorithmInterface类

第二步：实现AlgorithmInterface类的纯虚函数algorithm\_core

第三步：alg\_tool目录下，修改Makefile文件

第四步：编译生成so文件，可在lib目录下查看

第五步：conf/alg下修改或新增算法配置文件

## 6.5 更新本地数据资源

使用新的数据覆盖旧数据，然后发送命令即可

|  |
| --- |
| // 如下命令将更新所有的本地数据资源  python2.6 bin/binserverclient.py ${ip} ${control\_port} ‘update\_global’  // 如下命令将更新指定的本地数据资源  python2.6 bin/binserverclient.py ${ip} ${control\_port} ‘${global\_DB\_NAME}’ |

## 6.6 更新远程数据资源

使用新的配置文件覆盖conf/db\_config.ini

|  |
| --- |
| python2.6 bin/binserverclient.py ${ip} ${control\_port} ‘update\_db’ |

## 6.7 更新业务

更新业务需要执行如下命令：

|  |
| --- |
| python2.6 bin/binserverclient.py ${ip} ${control\_port} ‘update\_work’ |

根据更新方式不同，分述如下：

（1）新增一个业务

第一步：将业务的so文件发送到lib目录下；

第二步：在conf/work\_config.ini下添加新业务配置

第三步：执行命令

（2）删除一个业务

第一步：在conf/work\_config.ini中删除该业务配置

第二步：执行命令

（3）更新一个业务

该过程需要将业务先删除，再新增，否则容易出现程序终止的问题。

第一步：在conf/work\_config.ini中删除该业务配置

第二步：执行命令

第三步：将业务的so文件发送到lib目录下；

第四步：在conf/work\_config.ini下添加新业务配置

第五步：执行命令

bin/restartwork.sh是一个示例脚本

## 6.8 内存泄露测试

第一步：准备好测试case，/data/user\_list

第二步：修改src/lab\_common\_main.cpp，将代码63行的请求串，修改为业务的请求串

第三步：编译代码，生成bin/lab\_common\_main

第四步：执行bin目录下的memleak.sh

第五步：查看执行结果

（1）bin/log文件：保存了使用valgrind内存泄露的结果

（2）result目录：程序运行的结果

（3）log/lab\_common\_main.log：程序运行的日志

# 7 配置文件说明

## 附件1 db\_config.ini

|  |  |
| --- | --- |
| [DB\_INFO]  DBS\_SPLIT\_CHAR=44  DBS\_NAME= EXAMPLE\_REDIS\_DB  [EXAMPLE\_REDIS\_DB]  DB\_ID=0  DB\_TYPE=0  SVR\_IP=127.0.0.1,127.0.0.1;127.0.0.1,127.0.0.1  SVR\_PORT=6380  DB\_INDEX=0  TIMEOUT=20  SELECT\_TYPE=0  COMPRESS\_FLAG=0  IP\_TYPE=0  IP\_SPLIT\_FIRST\_CHAR=59  IP\_SPLIT\_SECOND\_CHAR=44  PORT\_SPLIT\_CHAR=44  KEY\_VALUE\_TYPE=0  VALUE\_STRING\_TYPE=0  VALUE\_SPLIT\_FIRST\_CHAR=44  VALUE\_SPLIT\_SECOND\_CHAR=  MGET\_TYPE=1  MGET\_THREAD\_NUM=100  USER\_PASSWD=  API\_PASSWD\_FLAG=0  API\_TOKEN\_FLAG=0 | 所有的db info信息  下面一行的分隔符  所有db的名称  DB名称  DB编号  DB类型  服务器ip组，每一组网通在前，电信在后  服务器端口  数据库索引，用于redis  访问超时时间  如何选择ip  结果是否压缩  ip类型，0表示网通，1表示电信  ip第一层分隔符，通常用分号  ip第二层分隔符，通常用逗号  端口分隔符  key的类型，目前统一使用0  value的字符串类型，目前统一使用0  结果的第一层分割符  结果的第二层分隔符  多个key时，是否拆分后并行获取  拆分步长  用户名和密码，冒号分割，用于http请求  是否需要密码  是否需要token |

注：

i. 关于DB\_TYPE，请参考include/db/db\_interface.h中DB\_TYPE枚举类型的定义

ii. 关于SELECT\_TYPE，请参考include/db/db\_interface.h中get\_ip\_port函数的定义

## 附件2 global\_db\_config.ini

|  |  |
| --- | --- |
| [GLOBAL\_DB\_INFO]  DBS\_SPLIT\_CHAR=44  DBS\_NAME=EXAMPLE\_SET\_DB,EXAMPLE\_KV\_DB  [EXAMPLE\_SET\_DB]  DB\_ID=0  DB\_TYPE=1  DB\_FILE\_NAME=../data/example\_set  KV\_SPLIT\_CHAR=9 | 所有的db info信息  下面一行的分隔符  所有db的名称  DB名称  DB编号  DB类型  数据文件位置  数据结构划分字符 |

注：关于DB\_TYPE，请参考include/db/global\_db\_interface.h中GLOBAL\_DB\_TYPE枚举类型

## 附件3 work\_conf.ini

|  |  |
| --- | --- |
| [WORK\_INFO]  WORK\_SPLIT\_CHAR=44  WORK\_NAMES=EXAMPLE\_WORK,SIMPLEFRAME\_WORK  CONF\_FOLDER=../conf/  LIB\_FOLDER=../lib/  LOG\_PATH=../log/lab\_common.log  [SIMPLEFRAME\_WORK]  API=simpleframe  SO\_NAME=SimpleFrameWorkInterface  ALG\_INI=alg/example\_alg.ini | 所有的业务信息  下面一行的分隔符  所有业务的名称  配置文件目录  库文件目录  日志文件目录  业务名称  业务请求json中的api名称  业务使用的库文件名称  业务使用的算法配置文件 |

## 附件4 lab\_common\_svr.conf

|  |  |
| --- | --- |
| log\_path : ../log/lab\_common.log  log\_buf\_size : 1024000  log\_type : ewidt  log\_flush\_type : ewi  ip : 0.0.0.0  query\_port : 21345  control\_port : 21346  thread\_num : 24  recv\_buf\_size : 1024000  send\_buf\_size : 1024000  recv\_timeout : 1000000  send\_timeout : 1000000  long\_conn : 1 | 日志存储地址  日志的buffer大小  日志级别  日志强制写入级别  本地ip地址  服务端口  控制端口  并发线程数  接收缓冲区的大小  发送缓冲区的大小  接收超时时间  发送超时时间  是否长链接 |

注：日志由强及弱分为ERROR, WARN, INFO, TRACE, DEBUG五种级别，可调用对应的LOG\_ERROR，LOG\_WARN，LOG\_INFO，LOG\_TRACE，LOG\_DEBUG函数。

log\_type：表示日志中会记录哪些级别的日志，使用各级别的首字母表示，不区分大小写。

log\_flush\_type：表示日志中哪些级别会强制写缓冲区

## 附件5 lab\_common\_main.conf

|  |  |
| --- | --- |
| log\_path : ../log/lab\_common\_main.log  log\_buf\_size : 1024000  log\_type : ewidt  log\_flush\_type : ewi  thread\_num : 4  output\_file\_path : ../result/output\_file  keys\_file\_path : ../data/user\_list | 日志存储地址  日志的buffer大小  日志级别  日志强制写入级别  并发线程数  输出文件地址  执行数据来源 |

## 附件6 example\_alg.ini

|  |  |
| --- | --- |
| [ALG\_INFO]  SO\_SPLIT\_CHAR=44  SO\_NAMES=INTER  SO\_FOLDER=../lib/  [INTER]  UID\_PLAN=0,InterAlgorithmInterface,|-1,InterAlgorithmInterface,|-2,InterAlgorithmInterface,WHITE\_USER\_LIST | 所有的算法信息  下面一行的分隔符  所有算法的名称  算法文件的目录  算法名称  算法策略：-2表示白名单；-1表示默认；其它数字表示使用尾号控制。每一个数字后面必须跟一个算法名称，-2后面还需指定白名单列表 |