PHP\_框架\_SwoftFramework框架(2.0.6)

--王尔贝(我是害虫)

--

^2\_3^2workroom3工作室(绝境求生)

--

2019-09-21 王尔贝(我是害虫)

--

//// SwoftFramework2.x(2.0.5)

https://www.swoft.org/docs/

//

//// Swoft项目

https://github.com/swoft-cloud/swoft

//

//// Swoole文档

https://wiki.swoole.com/wiki/page/965.html

--

--

第一篇、简介

--

--

一、框架特性

( https://www.swoft.org/docs/ )

--

1、概述

(图 https://raw.githubusercontent.com/swoft-cloud/swoft/master/public/image/start-http-server.jpg)

首个基于 Swoole 原生协程的新时代 PHP 高性能协程全栈框架，内置协程网络服务器及常用的协程客户端，常驻内存，不依赖传统的 PHP-FPM，全异步非阻塞 IO 实现，以类似于同步客户端的写法实现异步客户端的使用，没有复杂的异步回调，没有繁琐的 yield, 有类似 Go 语言的协程、灵活的注解、强大的全局依赖注入容器、完善的服务治理、灵活强大的 AOP、标准的 PSR 规范实现等等，可以用于构建高性能的Web系统、API、中间件、基础服务等等。

(1)、基于 Swoole 扩展；

(2)、内置协程网络服务器(Http/Websocket/RPC/TCP)；

(3)、高性能路由；

(4)、强大的 AOP (面向切面编程)；

(5)、灵活的注解功能；

(6)、全局的依赖注入容器；

(7)、基于 PSR-7 的 HTTP 消息实现；

(8)、基于 PSR-11 的容器规范实现；

(9)、基于 PSR-14 的事件管理器；

(10)、基于 PSR-15 的中间件；

(11)、基于 PSR-16 的缓存设计；

(12)、可扩展的高性能 RPC；

(13)、RESTful 支持；

(14)、国际化(i18n)支持；

(15)、快速灵活的参数验证器；

(16)、完善的服务治理，熔断、降级、负载、注册与发现；

(17)、通用连接池 Mysql、Redis、RPC；

(18)、数据库 ORM；

(19)、协程、异步任务投递；

(20)、自定义用户进程；

(21)、强大的日志系统；

项目地址：https://github.com/swoft-cloud/swoft；

--

--

二、关于 Swoft

( https://www.swoft.org/docs/2.x/zh-CN/introduction/swoft.html )

Swoft是基于Swoole Enterprise的最流行的PHP应用程序开发框架。有大量开发人员使用Swoft Framework来创建高性能，易于测试和可重用的代码。

Swoft 开源框架自2018年3月6号发布以来，一直在迭代升级，秉承简单、高效、稳定的宗旨。伴随着 1.x 的开发迭代和 Swoole 4 全协程化，1.x 的底层架构已经不再适合。从2018年11月开始规划 2.x, 底层借鉴 1.x 的经验全部重写，采用 Swoole hook 方式框架全部协程化，框架整体相比 1.x 更简单，稳定性也提升一个档次。

Swoft 是一款轻量级的框架，所有组件可以自定义，按需使用。Swoft 框架可以用于开发任何 Web 应用程序，构建高性能的 Web 系统、API、中间件、基础服务等等。

--

1、Swoft 优势

以下是使用Swoft Framework 的很多好处的列表。

(1)、以组件化方式开发，开发者可以自定义，按需加载使用。

(2)、框架是一个设计良好的 Web MVC 框架，它为 Web 框架提供了一个很好的选择。

(3)、为方便 PHP 开发者快速上手，数据库、缓存的使用都高度兼容 Laravel。

(4)、所有组件都严格加上单元测试和压力测试。

--

2、注解 (Annotation)

Swoft 是首个基于 Swoole 的注解框架，注解思想大量参考 spring boot, 在它之上进行简化，设计成更适合 phper 开发，使用更简单。

--

3、依赖注入（DI）

Swoft 最认同的技术是控制反转的依赖注入（DI）。该控制反转（IOC）是一个笼统的概念，它可以在许多不同的方式来表达。依赖注入仅仅是控制反转的一个具体例子。

在编写复杂的 PHP 应用程序时，应用程序类应尽可能独立于其他 PHP 类，以增加重用这些类的可能性，并在单元测试时独立于其他类测试它们。依赖注入有助于将这些类粘合在一起，同时保持它们独立。

完全依赖注入是什么？我们分别看这两个词。这里依赖部分转化为两个类之间的关联。例如，A类依赖于B类。现在，我们来看第二部分，注入。所有这些意思是，B类将被IoC注入A类。

依赖注入可以通过将参数传递给构造函数或通过使用setter方法进行后置构造来实现。由于依赖注入是 Swoft 框架的核心，我们将在相关示例的单独章节中解释这个概念。

--

4、连接池 (Pool)

依托于 Swoole 框架自带连接池，提供简单配置就可以使用，无需任何其它成本。

--

5、RPC

框架封装了一套高性能的 RPC 服务，每个 RPC 调用可以像本地函数一样使用，服务之间通过接口的方式约定规则，完全屏蔽实现细节，大大提升了开发效率，减少了维护成本。

--

6、面向切面编程（AOP）

Swoft的关键组件之一是面向切面编程（AOP）框架。跨越应用程序多个点的功能称为横切关注点，这些横切关注点在概念上与应用程序的业务逻辑分离。有各种常见的很好的例子，包括日志记录，声明式事务，安全性，缓存等等。

OOP中模块化的关键单元是类，而在AOP中，模块化的单元是切面。DI可帮助您将应用程序对象彼此分离，而AOP可帮助您将交叉关注与其影响的对象分离开来。

Swoft框架的AOP模块提供了一种面向切面的编程实现，允许您定义方法拦截器和切入点，以干净地分离实现应该分离的功能的代码。我们将在单独的章节中更多地讨论Swoft AOP概念。

--

--

三、版本更新 v2.0.6

( https://www.swoft.org/docs/2.x/zh-CN/introduction/update.html )

--

1、v2.0.6

(1)、修复(Fixed)

修复 ws server收到消息时，默认使用当前fd作为响应消息的客户端 3a8890cf

修复 ws server通过swoole server拿到的全部连接里，fd的类型可能不是整型导致报错。0fa79d7

修复 stdlib SystemHelper 在 windows 10 中会被 Defender 删除 71c2893

修复 increment/decrement 多拿了一个连接 94c7171

修复 模型属性 $modelTimestamps 关闭了无效 94c7171

修复 在初始化前调用了log，导致可能会在启动阶段报context丢失错误 5a27704f

修复 ws server的握手和open使用同一个顶级协程ID，如果有一个先执行完成，可能会导致后面的出现session丢失 c2cd17a

修复 after 定时器传递参数 bug 8ed609a

修复定时任务内存泄漏和日志打印数据丢失问题 aa6bc32

优化 Http server sgo 处理流程 e60d92e

修复数据库在事物情况下重连导致数据不一致问题 c2e87f2

修复 Aop 极端情况返回 0 问题 85bc1b1

修复

Factory::createBean() 方法创建问题c2e2c89

修复 RPC 错误数据是字符串问题 3a34658

//

(2)、更新(Update)

调整 ws server在worker 退出时自动关闭连接，添加开关配置，可以选择是否开启。3a8890c

ws server 新增事件： MESSAGE\_PUSH OPEN\_BEFORE MESSAGE\_RECEIVE e1b0583

//

(3)、增强(Enhancement)

新增日志按时间切割 f649df2

新增日志支持传递额外参数 55d5533

调整和新增了一些ws server相关事件，方便用户使用 e1b0583

模型对 Prop 支持 630aad8

新增连接连接池(mysql/redis/rpc)启动初始化连接功能 a367173

tcp server 使用长度分包时，支持自定义头部数据打包、解包格式 a6fbe932

//

(4)、扩展(Extra)

swoftcli 已经支持基本的应用骨架和组件结构创建，以及部分类文件生成功能

swoftTracker 支持方便开发者调试

--

2、v2.0.5-beta(2019-08-06)

(1)、升级提示

请使用 context() 代替 Context::mustGet()

这个版本开始验证器默认关闭，需要开发者开启 validator setting

//

(2)、修复(Fixed)

修复提前中断请求逻辑可能导致无法正确的格式化对应的Content-Type f031398

修复使用sgo创建子协程，在结束时没有清理与顶级协程的映射关系 de11ae5b

修复 Xml 格式解析方法 24d0038

修复类代理 Unique 字符串问题 444ddeb

修复 context()->get() 存在问题以及废弃 Context::mustGet() 方法 6acc1b5

修复接口注入，类名访问错误 2d9a31f

修复 模型使用 paginate 方法, 结果集没有映射字段3027287

修复 websocket server 里，当有多个worker，主动关闭非当前worker连接时报错 7666969

//

(3)、更新(Update)

默认的 worker num 设置通过 swoole\_cpu\_num 函数获取 553f6500

验证器调整成，默认非必须，用户自己开启 e5b258d

不再抛出swoft内部的 ContainerException 到用户使用层 1e74893

console output 新增 confirm select 等交互方法 0181138a

//

(4)、增强(Enhancement)

console 命令选项设置 CommandOption，现在允许选项描述为多行信息 e5914983

websocket server 的全部消息发送方法，支持传入 opcode 参数 dc164ffe9

websocket 模块允许设置当前模块的默认 opcode，用于自动处理返回数据设置 opcode 9e3e9672

增强 websocket 相关类，现在消息处理方法允许注入更多数据对象类型 Request Response 3d6c60b4

当worker退出时，websocket server 会自动关闭所有连接 5ea10ecc

Add Timer 定时器封装 eb92ee6

Add Coroutine 方法封装 d8c2b88

//

(5)、扩展(Extra)

swoft/crontab 秒级别定时任务

swoft/whoops 对 filp/whoops 在swoft中使用的简单封装，用于渲染并显示更加利于阅读的错误信息

--

3、v2.0.4-beta(2019-07-21)

(1)、升级注意

请去掉 bin/swoft 里的 Runtime::enanbleCoroutine() 设置

请确保 swoole 的 swoole.use\_shortname 的值为 On

从这个版本开始要求 swoole 4.4.1+

//

(2)、增强(Enhancement)

Swoft\Http\Message\Request 新增 getHeaderLines() (74a2a91)

Aop 新增 getArgsMap() 和 getClassName() 方法 (c47e785)

新增 srun() 函数，用于协程调度 (3c4a6a4)

优化 server 事件(onStart / onWorkStart / onWorkStop / onShutdown)，事件自带支持协程 (a8d5a8d)

新增投递同步阻塞任务(ec938e5)

新增 Redis call 方法, 用于使用同一连接操作(92456987)

兼容 Swoole 4.4.x

//

(3)、修复(Fixed)

修复 迁移类名太长导致记录类名不全(58314b8)

修复 实体查询之后使用Setter更新字段值之后update更新无效(caadf0e)

修复 stop 后删除pid文件的结果返回错误，导致restart失败 (2be450bf11)

修复 i18n 设置默认语言不生效的问题 (b401a504e)

修复 ws server在有多个worker时，无法主动关闭其他worker的连接(271b6398)

修复 http server接收xml请求时，content type 不能正确匹配(2ff9a4e61)

修复 使用 Database, json 操作无效(92456987)

修复 limiter 限速器 Redis 加前缀无法使用问题(7b54d4c)

//

(4)更新(Update)

更新 ws server 可以通过配置 disabledModules 来禁用 ws 模块(fa31111d)

//

(5)、扩展(Extra)

在官网增加案例展示，欢迎大家提交案例到官方案例仓库 swoft-cloud/swoft-case

在GitHub上对文档的修改，将会自动更新到官网文档，不再需要手动刷新

//

(6)、新增(New)

进程管理(c61b8f7)

自定义用户进程(c328d54)

进程池(d2e6290)

TCP server 基本可用

--

4、v2.0.3-beta(2019-07-08)

(1)、不兼容(Incompatible)

移除 request->json() 方法(c9e8f04) 替代方法 getParserdBody()

应用日志和控制台日志 levels 由数组配置改成，字符串配置，详细见文档

//

(2)、新增(Enhancement)

新增接口依赖注入(6169f84)

新增 getFile 方法获取文件上传保存之后的信息(fe7e3a6)

新增 restart() 服务新增重启方法(2ffec37)

新增调用 1.x RPC 服务支持(30d73c3)

新增 AOP 类名匹配支持正则表达式(bc5e479)

新增 RPC Server /Http Server 中间件命名空间 use 错误提示(b1cec04)

新增 验证器排除属性字段 unfields(b1bf44f)

新增 自动写入时间戳(dc58011)

新增 模型动作事件(dc58011)

新增 数据库迁移(26bb464)

新增 实体自动与 json 和数组互转(dc58011)

新增 模型批量更新方法batchUpdateByIds(dc58011)

//

(3)、修复(Fixed)

修复 cookies 设置时的一些问题，增加一些 withCookie 相关方法(b05afbb01)

修复 在console使用协程方式运行命令时，没有捕获处理错误(8a5418bf)

修复 websocket server 重启命令没有先停止旧server问题(db2d935)

修复任务返回值为 null 问题(a69347c)

修复 RPC Server 只有类中间件无法使用问题()204bc7f

修复 RPC Server 返回值为 null 问题(4d091be)

修复 Logger 和 CLog 日志等级无法覆盖和无效问题(8eb8aba)

修复 模型里面的属性不支持自定义表达式(dc58011)

//

(4)、更新(Update)

验证器优化，支持自定义验证规则(d959a4f)

重命名错误处理管理类 ErrorHanlders 为 ErrorManager (f3a8f04b)

console组件的异常处理改为由error组件提供的统一处理风格 (4f47204)

console组件允许设置禁用命令组(c5a0269)

在默认的错误处理中，允许设置错误捕获级别。默认级别是 E\_ALL | E\_STRICT (afff9029)

优化 启动ws server时同时启用了http处理功能，信息面板添加提示(83a81170)

优化 启动ws server 并同时添加rpc server启动，信息面板没有显示 rpc server信息(3d1d0d848)

//

(5)、扩展(Extra)

文档添加支持通过google进行搜索

新增 apollo 组件

新增 consul 组件

新增 breaker 组件

新增 limter 组件

新增 tcp-server 组件

--

5、v2.0.2-beta(2019-06-21)

(1)、新增(Enhancement)：

新增 $request->parsedQuery() 方法 (ab45089)

新增 Bean 属性注入基础数据类型(string/int/bool/float/array)，自动根据注释类型转换。(e3d4085)

新增 db(), 使 Model/Query/DB 支持切库(f3b12c9)

新增 DbSelectorInterface 用于根据业务自动切库(b36ca03)

新增 Http server 增加Cookies响应支持 (1a024bf)

新增 devtool 组件的实体生成(14b4d39c)

新增 允许注册 swoole 的 pipeMessage, packet 事件[afec0e3f]

新增 添加更多启动时的相关事件[8aaa38c4]

新增 validate() 不依赖注解验证方式(cc35560)

新增 RPC 调用返回错误 RpcResponseException 异常(baa482f)

新增 允许注册 swoole 的 pipeMessage, packet 事件(afec0e3f)

新增 添加更多启动时的相关事件(8aaa38c4)

//

(2)、修复(Fixed)

修复 paginate 无法指定查询字段(308d330)

修复 实体 join 操作，显示不出关联表数据 (446a3a2)

修复 实体属性名attributes等与系统定义属性冲突问题 (1a9d25b)

修复 AOP 切面重复执行问题 (54e00ac)

修复 Http 服务全局异常 contentType 失效问题(e704116)

修复 Http server在启用https 时，type 检查无法通过 (3e9b431)

修复 Http server通过data响应html时可能出现错误(e5513df)

修复 控制器 table 数据显示 (4d27718)

修复 devtool 生成的 phpstorm meta文件key重复(1838552)

修复 ws-server 参数路由匹配失败(4d3392e6)

修复 Http 服务 content-type 多值解析错误问题(1657979)

修复 Co::multi 抛出异常调用错误问题(d0ff0f7)

修复 body 字符串无解析调用错误问题(32abf60)

修复 通过 setter操作实体 toArray失效问题(9c0f9505)

现在停止server(包含Ctrl+C)时总是会删除pid文件，修复没有删除pid文件导致docker重启失败问题(6a96b2c)

修复 通过 @Inject 注入Redis连接池没有释放以及连接池全部空闲判断问题(feac3700)

修复 数据库开启事物多连接池操作错误(3f80474)

修复 init() 方法中使用 config() 函数问题(fe9b92b)

修复 getParsedQuery() 方法初始化问题(8e05836)

修复 一些bean会出现重复初始化(例如 config)问题 (f8ef5a9a)

修复 Autoloader 文件与其它composer包文件冲突问题 (ba19af363)

//

(3)、更新(Update)

修改模型方法insert,insertGetId,updateOrInsert只会插入定义了@Column字段(9c0f9505)

启动swoft时加入冲突扩展相关的运行环境检查(7fa7188f)

调整 count,sum,avg,max,min 返回值类型(c9b5e17a)

Http server 路由匹配顺序优化(d8fb588)

完善 RPC Client 调用错误信息提示(a1e56eb)

调整 Redis zAdd 成员和分数的位置⚠️不向下兼容(1a3275b3)

调整 如果没有添加验证注解，会跳过验证解析相关流程(582c806cc)

去除swoole中已经废弃的 buffer full/empty 事件相关代码 (873a97bed)

优化 Http server 验证器流程(0e73a8c)

//

(4)、扩展(Extra)

swoft/view 添加通过 @View 渲染视图支持

--

6、v2.0.1-beta (2019-05-29)

(1)、新增(Enhancement)

新增验证器不存在错误提示(5ba8682)

新增 AOP 代理类名称标识 \_PROXY\_(5ba8682)

新增实体字段值类型转错误异常提示(ab3cfea)

新增 BeanFactory::getBeans() 根据类名查询使用实例(999c10f)

新增 Model getAttributeValue 获取字段值方法(642d791e)

新增 Get 方式参数验证(6452a37)

新增 Redis 通过 Inject 方式注入使用(fc0a88e)

新增 RRC 服务全局异常处理(1653b24)

//

(2)、修复(Fixed)

修复 CLog 日志调用栈错误(868102f)

修复 Co::multi 调用失败错误数据格式(5ba8682)

修复 Function.php 重复加载(b063dd1)

修复 Logger 日志调用栈错误(ab3cfea)

修复 worker 进程退出未释放(close)数据库、缓存、RPC 连接池(5b1baa5)

修复 开发者自定义 content-type 失效问题(22cf706)

修复 request/session 级别销毁失效问题(e0d1493)

修复 bean 名称包含 . 点号失效问题(dc37c92)

修复连接池 maxIdleTime 参数无效问题(0e05591)

修复 文件上传 (10bc7fa)

修复 默认输出编码为UTF8 (755c8cf)

优化 Redis 配置类型转换(533b592)

打包为phar运行时路径错误(533b592)

//

(3)、Swoft-cli

修复一些问题 swoft-cli

发布新版本时自动通过travis部署phar到GitHub release (d1503b2)

//

(4)、Devtool

新 bean 方法代码智能提示(bean()/BeanFactory::getBean()/Swoft::getBean())(b63fe07)

--

7、v2.0.0-beta (2019-05-21)

Swoft 2 重构归来，基于 Swoole hook 方式，底层完全重构，相比之前更佳简单、稳定、易上手。

//// 功能特性

内置高性能网络服务器(Http/Websocket)

灵活的组件化

灵活的注解功能

多样化命令终端(Console)

强大的切面编程（AOP）

完善的容器管理、依赖注入 (DI)

灵活的事件机制

基于 PSR-7 的 HTTP 消息实现

基于 PSR-14 的事件管理器

基于 PSR-15 的中间件

可扩展的高性能 RPC

国际化(i18n)支持

简单高效的参数验证器

高性能连接池(Mysql/Redis/RPC)，自带断线重连

数据库高度兼容 laravel

缓存 Redis 高度兼容 Laravel

高效任务处理

灵活的异常处理

强大的日志系统

--

--

四、常见问题

( https://www.swoft.org/docs/2.x/zh-CN/introduction/question.html )

--

1、The HTTP server have been running!(PID: xx)

出现这种问题，是因为服务已经启动了。通常两种方式解决，第一种方式，stop 服务。第二种方式，kill pid。

--

2、Could not scan for classes inside xxx which does not appear to be a file nor a folder

出现这种问题是composer源的问题，一般会在创建项目(composer create swoft/swoft swoft)的时候出现，解决办法很简答只需要切换源全局设置即可。如：composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/。

更多源的切换请参阅 Composer配置( https://www.swoft.org/docs/2.x/zh-CN/ready/composer.html )。

--

--

五、参与 Swoft

( https://www.swoft.org/docs/2.x/zh-CN/introduction/join.html )

--

Swoft 开源框架，非商业项目，由开源组织开发与维护，这意味着任何人都可以为其开发和进步贡献力量。参与 Swoft 有多种方式：代码贡献、文档贡献、官网维护、社区维护。

--

1、代码贡献

Fork 组件代码库 swoft-component( https://github.com/swoft-cloud/swoft-component )；

发送 Pull Request 修改请求；

等待 Swoft 开发组审核和合并；

所有官方维护的组件均由swoft-component( https://github.com/swoft-cloud/swoft-component )项目统一合并后分发到各个组件。

--

2、参与 swoft 的开发是否很难

通过对比 swoft 和传统框架, 来回答这个问题。

(1)、框架定位

传统框架: 适合开发「巨石应用」, 所有功能都包含在同一份代码库中, 适合初创团队快速实现原型；

swoft: 致力于实现 PHP 下的微服务, 功能都由「组件」提供, 可以按照实际需求按需加载组装；

//

(2)、开发难度 (难? 我亦无他, 唯手熟尔。)

A、传统框架

熟悉经典的 lnmp 体系, 只用考虑 PHP 语法；通常需要完整熟悉整个框架, 才可能参与开发, 老旧的框架提 PR 可能很久才会处理；

B、swoft

熟悉 swoole 以及网络编程的基础知识, 就能轻松用 PHP 写出并发能力远超 fpm；开发组提供开发指南, 组件说明, 快速响应, 开发者可以选取一个简单的组件上手, 快速完成一个 PR；

--

3、开发步骤

(1)、使用 docker

推荐使用 docker 解决开发环境的困扰, 开发组维护了 swoft/swoft ( https://github.com/swoft-cloud/swoft ) 和 swoft/alphp 两个镜像, 简单示例。

A、使用 docker-compose 进行服务编排

version: '3.1'

services: # 定义服务

    swoft: # 服务的名称

        image: swoft/alphp:cli

        volumes:

            - ../:/data # 挂载本地的源码到容器的 /data 目录下

        ports:

            - "80:9501" # 映射本地 80 端口到容器 9501 端口

        links: # 其他基础服务也可以用 docker 解决, 用不到可以先去掉

            - redis

            - mysql

            - rabbitmq

        tty: true # 开启后, 可以使用 exec 进入到容器中

//

B、简单使用

docker-compose up -d swoft # 启动服务

docker-compose exec swoft # 进入容器中

//

(2)、swoft 项目说明

//// swoft 包含2个重要项目

swoft-cloud/swoft-component( https://github.com/swoft-cloud/swoft-component ): swoft 源码, 组件都在此项目的 src/ 目录下。

swoft-cloud/swoft( https://github.com/swoft-cloud/swoft ): swoft 脚手架, 加载了全部组件并提供了所有组件的使用 demo。

//

//// 换种说法

参与 swoft, 其实在做应用层的事, 业务中要怎么写, 参考 swoft 中的 demo 就行, 通俗说 -- 造飞机。

参与 swoft-component, 其实是参与其中具体的某个/某几个组件, 通俗说 -- 造飞机引擎。

//

//// swoft 的 架构设计, 体现在 swoft 的 组件化 实现中

框架组件 framework: 框架核心工作, 以及框架核心配套的组件。

swoft 功能组件: 按照功能进行划分, 依赖框架组件, 可以按需加载, 常见的有 rpc 全家桶: rpc rpc-server rpc-client；cli 引用: console。

//

如果只是想 造飞机, 只用给 swoft 项目提 PR 即可; 如果想 造飞机引擎, 需要给 swoft-component 提 PR, 本地还需要一份类似 swoft 项目的脚手架, 用来验证。

//

(3)、举个例子: 参与 RPC 组件开发

如果对 RPC 的基本概念不熟悉, 参考 RPC 组件相关文档。新建一个空文件夹, 比如 swoft-test, 充当 swoft 项目的角色, 作为脚手架, 加载需要的组件。

//

///1/ 需要的基础文件

├── app

│   ├── Application.php

│   └── bean.php

├── bin

│   ├── bootstrap.php

│   └── test.php

├── composer.json

├── config

是的, 只需要这么多, 从 swoft 复制过来即可。

//

///2/ 使用 composer 加载 swoft 组件

{

  "require": {

    "swoft/rpc": "dev-master",

    "swoft/rpc-server": "dev-master",

    "swoft/rpc-client": "dev-master"

  },

  "autoload": {

    "psr-4": {

      "App\": "app/"

    },

  },

  "minimum-stability": "dev",

  "repositories": {

    "hyperf": {

      "type": "path",

      "url": "../swoft-component/src/\*"

    },

    "packagist": {

      "type": "composer",

      "url": "https://mirrors.aliyun.com/composer/"

    }

  }

}

这里涉及 composer 从本地加载依赖文件的用法, 不熟悉可以参考 官方文档 composer-repository( https://getcomposer.org/doc/04-schema.md#repositories )。

好了, 更新: composer u --no-dev。

之后就可以执行 php bin/swoft 来进行相应测试了, 如果遇到报错, 根据报错添加缺少的组件即可。

//

///3/ 修改 RPC 相关文件

修改 swoft-component 下的组件, 建议多和开发组交流, 有开发范式和 CI 来保证组件的开发质量和稳定性。

--

4、需要了解的更多知识

(1)、docker 基础知识；

(2)、github 如何提 PR。推荐一个好用的工具, github desktop, 有快捷键快速提 PR。

(3)、composer 文档, 组件开发和生产项目有所差异, composer 提供了一些有意思的功能, 方便开发组件。

推荐使用 mac + phpstorm + docker，构建全套开发环境，有相关问题，欢迎和开发组交流。

--

--

第二篇 开始

--

--

一、快速起步

( https://www.swoft.org/docs/2.x/zh-CN/quick-start/index.html )

--

0-1、环境搭建

接下来的几篇文章主要介绍swoft需要的运行环境搭建，以及一些使用注意。

(1)、运行swoft需要哪些环境的支持

(2)、在本机手动运行(只支持Mac Linux)

(3)、安装必须的包

(4)、使用docker运行

(5)、使用官方镜像

(6)、使用docker-compose

--

0-2、环境需求

(1)、必须安装的

安装PHP并且版本至少 >7.1；

安装php包管理器 composer；

连接迭代器依赖 pcre 库；

安装php扩展swoole, 并且版本至少 >4.3.0；

其他需要安装和启用的php扩展有：PDO redis；

//

(2)、有冲突的

下面列出一些已知的和swoole有冲突的php扩展，请使用swoft时不要安装或禁用它们：xdebug、xhprof、blackfire、zend、trace、uopz。

--

1、Annotations插件 (开发准备 PHPStorm 安装Annotations插件)

( https://www.swoft.org/docs/2.x/zh-CN/quick-start/development.html )

通过文本编辑器的环境进行 Swoft 开发，在使用 Swoft 便捷的注解时，仍需要 use 注解相对应的命名空间， 这显然不是一个高效的做法。

我们推荐在 PHPStorm 环境下，并安装 Jetbrain 自带的 PHP Annotations 插件，可提供注解命名空间自动补全，注解属性代码提醒，注解类跳转等非常有助于提升开发效率的功能。

--

2、安装Swoft

( https://www.swoft.org/docs/2.x/zh-CN/quick-start/install.html )

(1)、Docker方式安装（推荐）

docker run -p 18306:18306 --name swoft swoft/swoft

//

(2)、Docker-Compose安装（推荐）

git clone https://github.com/swoft-cloud/swoft

cd swoft

docker-compose up

///1/ 提示

docker-compose up 启动后，默认启动了swoft http server，并且它是容器的的第一个进程(pid=1)，因此进入容器里是 没法停止/重启server 的。

如果你想在启动后进入容器，手动管理swoft server的启动。需要先稍微调整下 docker-compose.yml。

swoft:

    image: swoft/swoft

#for local develop

    command: php -S 127.0.0.1:13300

添加一个commad命令覆盖默认的启动server命令。启动后，进入容器再手动启动server php bin/swoft http:start。此时，server的进程不是容器的第一个进程，就可以使用swoft server的其他命令进行停止/重启操作了。

//

(3)、Composer安装

composer create-project swoft/swoft swoft

通过 Packagist国内镜像 加速国内下载速度，请参阅 Composer配置( https://www.swoft.org/docs/2.x/zh-CN/ready/composer.html )。

//

//

//// 问题

//问题1/ 描述

Your requirements could not be resolved to an installable set of packages.

Problem 1

* The requested PHP extension ext-redis \* is missing from your system. Install or enable PHP's redis extension.

//

//问题1/ 解决 (环境php7.2.22/Redis4.0.14)

(php的官方扩展库--pecl https://pecl.php.net/ https://pecl.php.net/package/redis)因为除了用自己编译的办法安装redis扩展，还可以使用mac自带的pecl安装(就是类似于brew)。首先看到successfully证明安装成功，然后看扩展 redis.so安装的位置 ，再进入你的php.ini文件更改(当下为“/usr/local/etc/php/7.2/php.ini”)。

//

pecl install redis

从中读取出“/usr/local/Cellar/php@7.2/7.2.22\_1/pecl/20170718/redis.so”就行；

//

//问题2/ 描述

PHP Fatal error:  Uncaught Error: Class 'Swoole\Coroutine' not found in /Users/cenyebao/web/project-swoft2x-swooleframework/bin/swoft:7

//

//问题2/ 解决

[Mac]\_安装PHP扩展swoole；

// 安装wget

brew install wget

//

|1>、下载swoole源码，https://github.com/swoole/swoole-src/releases；

// 下载4.4.6

wget https://github.com/swoole/swoole-src/archive/v4.4.6.tar.gz

//

|2>、tar -zxvf swoole.tgz；

tar -zxvf v4.4.6.tar.gz

//

|3>、进入解压目录，输入"/usr/local/Cellar/php@7.2/7.2.22\_1/bin/phpize"(输入‘find . -name "phpize"’语句搜索)；

|4>、./configure --with-php-config=/usr/local/Cellar/php@7.2/7.2.22\_1/bin/php-config；

|5>、make && make install；

Build complete.

Don't forget to run 'make test'.

Installing shared extensions:     /usr/local/Cellar/php@7.2/7.2.22\_1/pecl/20170718/

Installing header files:          /usr/local/Cellar/php@7.2/7.2.22\_1/include/php/

|6>、修改php.ini，extension=swoole.so；

//

(4)、手动安装

git clone https://github.com/swoft-cloud/swoft

cd swoft

composer install --no-dev # 不安装 dev 依赖会更快一些

cp .env.example .env

vim .env # 根据需要调整启动参数

--

3、运行服务 (服务启动与管理)

( https://www.swoft.org/docs/2.x/zh-CN/quick-start/start-swoft.html )

Swoft 拥有完善的命令行工具，和相应的服务器管理命令。

//

(1)、帮助命令

[root@swoft]# php bin/swoft -h

Console application description (Version: 2.0.0)

Usage:

  bin/swoft COMMAND [arg0 arg1 arg2 ...] [--opt -v -h ...]

Options:

      --debug      Setting the application runtime debug level(0 - 4)

      --no-color   Disable color/ANSI for message output

  -h, --help       Display this help message

  -V, --version    Show application version information

      --expand     Expand sub-commands for all command groups

Available Commands:

  http     Provide some commands to manage the swoft HTTP Server(alias: httpserver,httpServer,http-server)

  rpc      Class ServiceServerCommand

  test     Class TestCommand

  ws       Provide some commands to operate swoft WebSocket Server(alias: ws-server,wsserver,websocket)

More command information, please use: bin/swoft COMMAND -h

reload只是重新释放一下内存，只在开发过程中有用，线上请勿使用。更新代码时可以放心使用restart，swoole底层做了优化，不会中断正在处理的请求。

//

(2)、HTTP 服务器

// 启动服务

php bin/swoft http:start

//

// 守护进程启动

php bin/swoft http:start -d

//

// 重启

php bin/swoft http:restart

//

// 重新加载

php bin/swoft http:reload

//

// 关闭服务

php bin/swoft http:stop

//

(3)、WebSocket服务器

启动WebSocket服务器,可选是否 同时支持http处理。

// 启动服务

php bin/swoft ws:start

//

// 守护进程启动

php bin/swoft ws:start -d

//

// 重启

php bin/swoft ws:restart

//

// 重新加载

php bin/swoft ws:reload

//

// 关闭服务

php bin/swoft ws:stop

//

(4)、RPC 服务器

使用独立的RPC服务器。

// 启动服务

php bin/swoft rpc:start

//

// 守护进程启动

php bin/swoft rpc:start -d

//

// 重启

php bin/swoft rpc:restart

//

// 重新加载

php bin/swoft rpc:reload

//

// 关闭服务

php bin/swoft rpc:stop

//

(5)、启动信息

默认情况下，启动时会打印一些启动信息到控制台。

如果你的 .env 开启了 SWOFT\_DEBUG=1 将会看到更多详细的启动与加载信息。如果你想关闭这些信息，可以在 app/Application 添加如下设置。

public function getCLoggerConfig(): array

{

    $config = parent::getCLoggerConfig();

    // disable print console start log

    $config['enable'] = false;

    return $config;

}

重启swoft，可以看到不会有任何信息输出了。

--

4、应用结构

( https://www.swoft.org/docs/2.x/zh-CN/quick-start/project-skeleton.html )

一个完整的swoft应用可以包含：console 应用、http 服务(跟传统的框架差不多)、websocket 服务、rpc 服务。swoft-cloud/swoft 即是一个完整应用的demo。当然，如果你只想使用一部分功能也是可以的。

//

(1)、应用骨架

app 下的类目录为了避免一些文件夹名称没有复数单词，导致命名不统一，所有的文件夹名称 统一使用单数。

//

├── app/   ------ 应用代码目录

│   ├── Annotation/   ------- 定义注解相关

│   ├── Aspect/       ------- AOP 切面

│   ├── Bean/         ------- 一些具有独立功能的class bean

│   ├── Console/      ------ 命令行代码目录

│   │   ├── Command/

│   ├── Exception/      ------ 定义异常类目录

│   │   └── Handler/     ------ 定义异常处理类目录

│   ├── Http/         ------ HTTP 代码目录

│   │   ├── Controller/

│   │   └── Middleware/

│   ├── Helper/

│   │   └── Functions.php

│   ├── Listener/     ------ 事件监听器目录

│   ├── Model/        ------ 模型、逻辑等代码目录(这些层并不限定，根据需要使用)

│   │   ├── Dao/

│   │   ├── Data/

│   │   ├── Logic/

│   │   └── Entity/

│   ├── Rpc/          ------ RPC 代码目录

│   │   └── Service/

│   │   └── Middleware/

│   ├── WebSocket/     ------ WebSocket 代码目录

│   │   ├── Chat/

│   │   ├── Middleware/

│   │   └── ChatModule.php

│   ├── Application.php -------- 应用类文件继承自swoft核心

│   ├── AutoLoader.php  -------- 项目扫描等信息(应用本身也算是一个组件)

│   └── bean.php

├── bin/

│   ├── bootstrap.php

│   └── swoft   ------ swoft 入口文件

├── config/     ------ 应用配置目录

│   ├── base.php  --- 基础配置

│   └── db.php

├── public/     ------ WEB可访问目录

├── resource/   ------ 应用相关资源目录

│   ├── language/   ------ 语言资源目录

│   └── view/       ------ 视图资源目录

├── runtime/    ------ 临时文件目录(日志、上传文件、文件缓存等)

├── test/       ------ 单元测试代码目录

│   └── bootstrap.php

├── composer.json

├── composer.lock

├── phar.build.inc

└── phpunit.xml.dist

//

render by tree -L 2 -F --dirsfirst

//

(2)、组件骨架结构

├── src/

│   ├── Annotation/  -------- 组件注解类定义

│   ├── Bean/         ------- 一些具有独立功能的 class bean

│   ├── Concern/      ------- traits classes

│   ├── Contract/     ------- interface classes

│   ├── Exception/

│   ├── Helper/

│   ├── Listener/

│   ├── AutoLoader.php  -------- 组件扫描等信息

├── test/   ------ 单元测试代码目录

│   ├── unit/

│   ├── testing/

│   └── bootstrap.php

├── LICENSE

├── README.md

├── composer.json

└── phpunit.xm

//

src/AutoLoader.php 是一个组件必须存在的文件，swoft依据它来确定要扫描哪些目录。

//

///1/ 开发自定义组件

在2.0版本你可以非常容易的开发一个swoft组件。延伸阅读 开发自定义组件( https://www.swoft.org/docs/2.x/zh-CN/component/index.html )。

--

二、技术说明

--

0-1、基础准备

( https://www.swoft.org/docs/2.x/zh-CN/ready/index.html )

在使用 swoft 之前你需要具备一些服务端编程基础理论，在传统的php-fpm开发模式下 你基本不需要担心，资源释放，共享变量等相关的问题。

不过你不用担心，在 2.x 版本，你可以像在php-fpm 模式下开发一样简单。如果你有用过 Laravel 入手就更快了，因为 DataBase 、 Redis 和在 Laravel 下使用基本一致。

--

0-2、入门须知

在 swoole 常驻内存开发模式中，不要用超全局变量共享数据，你可以使用 Context 代替，不要使用 curl 扩展，当然 guzzlehttp 底层也是 curl 驱动的，curl 暂不支持协程化，具体原因你可以参考swoole 官方文档( https://wiki.swoole.com/wiki/page/965.html )。

你可以使用 swoft 自带的 httpClient ，它和 guzzlehttp 一样方便快捷，还原生支持并发调用。你再也不用担心因为某个http请求时间过长从而导致服务器CPU 疯涨。不过在接下来 swoole 会支持curl 扩展，相信不会等待太久。当然还有很多细节在文档中有仔细描述，在这就不一一描述了。

--

1、传统架构

( https://www.swoft.org/docs/2.x/zh-CN/ready/tradition.html )

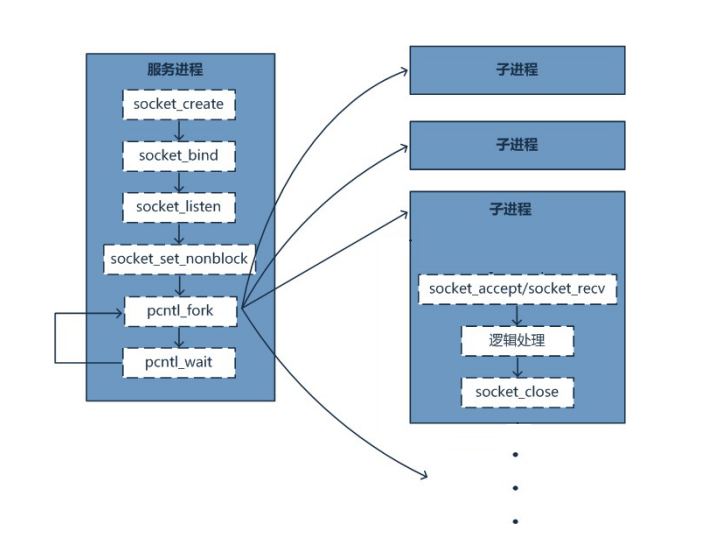
在 Nginx + PHP-Fpm 模式下开发非常简单 不用担心 内存泄露。php的fastcgi进程管理器php-fpm和nginx的配合已经运行得足够好，但是由于php-fpm本身是同步阻塞进程模型，在请求结束后释放所有的资源（包括框架初始化创建的一系列对象，导致PHP进程空转（创建<-->销毁<-->创建）消耗大量的CPU资源，从而导致单机的吞吐能力有限。请求夯住，会导致 CPU 不能释放资源， 大大浪费了 CPU 使用率。

//

(1)、php-fpm

php-fpm进程模型也非常简单的，是 属于预派生子进程模式 大家想必知道早期的 Apache 就是采用该模式 来一个请求就 fork 一个进程，进程的开销是非常大的。这会大大降低吞吐率，并发数由进程数决定。

(图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ready/fpm.png)



//

///1/ 预派生子进程模式

程序启动后就会创建N个进程。每个子进程进入Accept，等待新的连接进入。当客户端连接到服务器时，其中一个子进程会被唤醒，开始处理客户端请求，并且不再接受新的TCP连接。当此连接关闭时，子进程会释放，重新进入Accept，参与处理新的连接。

//

///1-1/ 优势

这个模型的优势是完全可以复用进程，不需要太多的上下文切换。

//

///1-2/ 劣势

A、这种模型严重依赖进程的数量解决并发问题，一个客户端连接就需要占用一个进程，工作进程的数量有多少，并发处理能力就有多少。操作系统可以创建的进程数量是有限的。

B、PHP框架初始化会占用大量的计算资源，每个请求都需要初始化。

C、启动大量进程会带来额外的进程调度消耗。数百个进程时可能进程上下文切换调度消耗占CPU不到1%可以忽略不计，如果启动数千甚至数万个进程，消耗就会直线上升。调度消耗可能占到 CPU 的百分之几十甚至 100%。

D、如果请求一个第三方请求非常慢，请求过程中会一直占用 CPU 资源，浪费了昂贵的硬件资源。例如：即时聊天程序，一台服务器可能要维持数十万的连接，那么就要启动数十万的进程来持，这显然不可能。有没有一种技术可以在一个进程内处理所有并发IO呢？答案是有，这就是IO复用( https://www.swoft.org/docs/2.x/zh-CN/ready/io.html )技术。

//

(2)、php-fpm工作模式的问题

nginx基于epoll事件模型，一个worker同时可处理多个请求；

fpm-worker在同一时刻可处理一个请求；

fpm-worker每次处理请求前需要重新初始化mvc框架，然后再释放资源；

高并发请求时，fpm-worker不够用，nginx直接响应502；

fpm-worker进程间切换消耗大；

//

(3)、那我们还有什么解决方案？

我们分析我们的业务不难发现，90%以上的业务都是IO密集性业务，我们只需要提高IO复用的能力就可以提升单机吞吐能力，另外需要将php-fpm同步阻塞模式替换为异步非阻塞模式，异步开启模式比较复杂不易维护，当然不一定使用php-fpm，就可以解决我们的核心问题——性能。

在 IO密集性业务 中我们需要频繁的上下文切换，采用线程模式开发太过复杂。

一个进程中能开的线程数也有限，线程太多也会增加 CPU 的负荷和内存资源，线程没有阻塞态，IO 阻塞也不能主动让出 CPU资源，属于抢占式调度模型。不太适合 php 开发。

在 swoole 4.+开启了全协程模式，让同步代码异步执行。详情请看为什么要使用 swoole( https://www.swoft.org/docs/2.x/zh-CN/ready/swoole.html )。

--

2、IO复用 (IO 多路复用)

这的复用指的是复用的线程，其实IO复用的历史和多进程一样长。

//

(1)、select/poll

// select

Linux很早就提供了 select 系统调用，可以在一个进程内维持1024个连接。

// poll

后来又加入了poll系统调用，poll做了一些改进，解决了 1024 限制的问题，可以维持任意数量的连接。

//

但select/poll还有一个问题就是，它需要循环检测连接是否有事件。这样问题就来了，如果服务器有100万个连接，在某一时间只有一个连接向服务器发送了数据，select/poll需要做循环100万次，其中只有1次是命中的，剩下的99万9999次都是无效的，白白浪费了CPU资源。

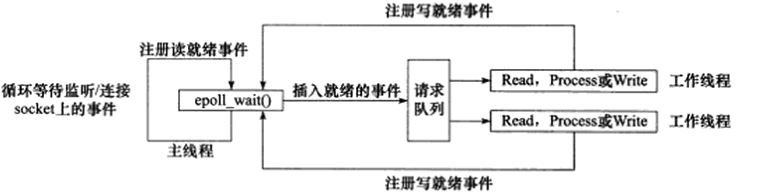
//

(2)、epoll

直到Linux 2.6内核提供了新的epoll系统调用，可以维持无限数量的连接，而且无需轮询，这才真正解决了 C10K 问题。现在各种高并发异步IO的服务器程序都是基于epoll实现的，比如Nginx、Node.js、Erlang、Golang。像 Node.js，Redis 这样单进程单线程的程序，都可以维持超过1百万TCP连接，全部归功于epoll技术。

在这就不得不提，基于 epoll 实现的 Reactor 模型，IO复用异步非阻塞程序使用经典的Reactor模型，Reactor顾名思义就是反应堆的意思，它本身不处理任何数据收发。只是可以监视一个socket句柄的事件变化。

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ready/eventloop.png )



//

///1/ 高效的事件处理模式Reactor模式

|1>、主进程/线程往epoll内核事件中注册socket上的读就绪事件。

|2>、主进程/线程调用epoll\_wait等待socket上有数据可读。

|3>、当socket上有数据可读时，epoll\_wait通知主进程/线程。主进程/线程则将socket可读事件放入请求队列。

|4>、睡眠在请求队列上的某个工作线程被唤醒，它从socket读取数据，并处理客户请求， 然后往epoll内核事件表中注册该socket上的写就绪事件。

|5>、主线程调用epoll\_wait等待socket可写。

|6>、当socket可写时，epoll\_wait通知主进程/线程主进程/线程将socket可写事件放入请求队列。

|7>、睡眠在请求队列上的某个工作线程被唤醒，它往socket上写入服务器处理客户请求。

swoole 的 Reactor 线程 也是基于 Reactor 模型实现的。

--

3、Swoole 扩展

( https://www.swoft.org/docs/2.x/zh-CN/ready/swoole.html )

(1)、为什么要使用 swoole？

|1>、swoole的强大之处就在于其进程模型的设计，既解决了异步问题，又解决了并行。

|2>、从4.0版本开始Swoole提供了完整的协程(Coroutine)+通道(Channel)特性，带来全的CSP编程模型。应用层可使用完全同步的编程方式，底层自动实现异步IO。

|3>、使用常驻内存模式 可以避免每次框架的初始化，节约了性能开销。 4.x 底层采用自动化协程转换。

//

(2)、什么是协程

Coroutine 是历史比线程的历史还要悠久，协程可以理解为纯用户态的线程，其通过协作而不是抢占来进行切换。相对于进程或者线程，协程所有的操作都可以在用户态完成，创建和切换的消耗更低。Swoole可以为每一个请求创建对应的协程，根据IO的状态来合理的调度协程。

开发者可以无感知的用同步的代码编写方式达到异步IO的效果和性能，避免了传统异步回调所带来的离散的代码逻辑和陷入多层回调中导致代码无法维护。

同时由于底层封装了协程，所以对比传统的PHP层协程框架，开发者不需要使用yield关键词来标识一个协程IO操作，所以不再需要对yield的语义进行深入理解以及对每一级的调用都修改为yield，这极大的提高了开发效率。

//

///1/ 多进程/多线程/协程对比

(选项 多进程 多线程 协程)

创建  fork  pthread\_create  go

回收  wait  pthread\_join  -

通信方式  IPC 进程间通  数据同步/锁  array/chan

资源消耗  进程切换开销  进程切换开销  非常低

并发能力  数百  数千  50万

编程难度  困难  非常困难  容易

//

(3)、协程的优点

用户态线程，遇到 IO 主动让出。

PHP 代码依然是串行执行的，无需加锁。

开销极低，仅占用内存，不存在进程/线程切换开销。

并发量大，单个进程可开启 50W 个协程。

随时随地，只要你想并发，就调用 go 创建新协程。

//

(4)、swoole 的协程

swoole 的协程 和 golang的调度方式完全不同，每一个进程里面的协程都是串行执行所以无需担心访问资源加锁问题，这也符合 php 简单的特性(那么 进程 的协程是串行 执行如何利用 多核 CPU 实现并行呢。答案是利用多进程实现。现在 task 也可以开启协程)。

这也许没有 golang 性能那么好，但是对于 IO 密集型业务是非常适合的，协程的上下文切换非常快。切换一门语言的成本对公司来说也是非常巨大的，现在 swoole 的生态也越来越好。

4.0 底层加入 Hook 机制，使原生的 Mysql PDO、Redis 操作协程化，后续将支持 Curl 扩展。更贴近传统业务代码，迁移成本也降低。

swoole 的 http server 采用优秀的 Reactor 模型，处理速度可逼进 NGINX 处理静态页面的 速度。对于 Api或者基础服务来说非常的适合。性能瞬间翻几翻，再也不用担心 php-fpm 进程数过多，导致 CPU 被打满。

//

(5)、swoole 需要注意的地方

当然它也不是没有缺点的。

|1>、无法做密集计算。当然这一点是php甚至是所有动态语言都存在的问题。写在这里是因为防止误导读者以为使用swoole后，php可以用来做密集计算。

|2>、更容易内存泄露。在处理全局变量，静态变量的时候一定要小心，这种不会被GC清理的变量会存在整个生命周期中，如果没有正确的处理，很容易消耗完所有的内存。而以往的php-fpm下，php代码执行完内存就会被完全释放。(更容易内存泄露。在处理全局变量，静态变量的时候一定要小心，这种不会被GC清理的变量会存在整个生命周期中，如果没有正确的处理，很容易消耗完所有的内存。而以往的php-fpm下，php代码执行完内存就会被完全释放)这个"通信"你可以理解利用channel通信。

|3>、虽然 swoole 协程是串行，但是业务有可能会交叉，比如你一个业务在添加配置一个业务在写配置，上下文一切换，可能会造成配置的不一致，你可能会好奇，本地跑跑着没事到了线上跑一段时间会出问题，这本是业务设计上的问题。

--

4、Composer (依赖管理工具)

( https://www.swoft.org/docs/2.x/zh-CN/ready/composer.html )

Composer 作为依赖包管理工具，使用前请确保您已安装PHP 和 Composer( https://getcomposer.org/download/ )。

//

(1)、使用国内源

aliyun(推荐) https://mirrors.aliyun.com/composer/

tencent https://mirrors.cloud.tencent.com/composer/

huaweicloud https://mirrors.huaweicloud.com/repository/php/

cnpkg.org https://php.cnpkg.org

laravel-china https://packagist.laravel-china.org

//

///1/ 方法1：修改全局配置

// 打开终端并执行如下命令

composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/

// OR

composer config -g repo.packagist composer https://mirrors.huaweicloud.com/repository/php/

//

///2/ 方法2：修改项目配置

打开终端，进入你的项目的根目录（也就是 composer.json 文件所在目录），执行如下命令。

composer config repo.packagist composer https://mirrors.aliyun.com/composer/

// OR

composer config repo.packagist composer https://mirrors.huaweicloud.com/reposi

您也可以手工在composer.json中添加如下内容。

"repositories": {

    "packagist": {

        "type": "composer",

        "url": "https://mirrors.aliyun.com/composer/"

    }

}

//

(2)、相关网址

PHP官方地址：http://php.net/

Composer官方地址：https://getcomposer.org/

--

--

三、特别注意

( https://www.swoft.org/docs/2.x/zh-CN/notice/index.html )

--

1、严禁使用

( https://www.swoft.org/docs/2.x/zh-CN/notice/prohibited.html )

禁止die()、exit()函数。

禁止使用$\_GET、$\_POST、$GLOBALS、$\_SERVER、$\_FILES、$\_COOKIE、$\_SESSION、$\_REQUEST、$\_ENV等超全局变量。

谨慎使用global、static关键字。

--

2、内存泄漏排查

( https://www.swoft.org/docs/2.x/zh-CN/notice/memory-leak.html )

--

3、性能调优

( https://www.swoft.org/docs/2.x/zh-CN/notice/performance-optimization.html )

--

--

第三篇 基础

--

--

一、注解

--

0、注解

( https://www.swoft.org/docs/2.x/zh-CN/annotation/index.html )

(1)、安装

// 安装注解

composer require swoft/annotation

//

(2)、什么是注解

注解其实就 PHP 代码里面的注释，但是它是有功能含义的。

//

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

use Swoft\Bean\Annotation\Mapping\Inject;

/\*\*

 \* 类注释

 \*

 \* @since 2.0

 \*

 \* @Controller("a")

 \*/

class A

{

    /\*\*

     \* 属性注释

     \*

     \* @Inject()

     \*

     \* @var string

     \*/

    private $pro = '';

    /\*\*

     \* @RequestMapping()

     \*

     \* 方法注释

     \*/

    public function method(): void

    {

    }

}

//

@Controller 注释，定义一个控制器，后续章节详细介绍。

@Inject 注释，定义一个依赖，后续章节详细介绍。

@RequestMapping 注释，定义一个 action，后续章节详细介绍。

严重警告⚠️ 注解必须以 /\*\*开始，并以 \*/ 结束。否则会导致无法解析！

//

(3)、原理(未更新)

//

(4)、规范

PHP 代码里面本身就会有很多注释，开发者写的有功能的注解，如果随意乱写，会导致不好维护。Swoft 框架源码，注解规范如下。

类注解，所有类注释后面。

属性注解，属性描述之后，其它注释之前。

方法注解，方法描述之后，其它注释之前。

Swoft 框架里面的注解规范仅供参考，如果有更好的方式，欢迎讨论。

//

(5)、Git仓库

Github https://github.com/swoft-cloud/swoft-annotation

//

(6)、参与贡献

欢迎参与贡献，您可以如此。

fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )。

修改代码然后发起 PR。

关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、为什么使用注解

( https://www.swoft.org/docs/2.x/zh-CN/annotation/why.html )

Swoft 框架使用注解最大的原因就是因为简洁、灵活。

--

2、开发工具

( https://www.swoft.org/docs/2.x/zh-CN/annotation/tool.html )

PHP 使用注解需要一个插件，否则非常不友好。

(1)、PhpStorm

安装 PHP annotations 插件，可以像写代码一样，写功能注解。它提供强大的智能提示，错误提示功能。

// 安装PhpStorm "PHP Annotations"插件

[MacbookPro]\_PhpStorm-Preferences-"Plugins-Brawse repositories.."-搜索"PHP Annotations"-安装；

--

3、如何使用

( https://www.swoft.org/docs/2.x/zh-CN/annotation/usage.html )

//

(1)、使用

注解使用很简单，和使用一个类一样，首先使用 use 关键字引入类，其次使用注解。

use Swoft\Http\Server\Annotation\Mapping\Controller;

/\*\*

 \* 类注释

 \*

 \* @since 2.0

 \*

 \* @Controller("a")

 \*/

class A

{

}

第1行引入注解类，像一个普通类一样。

第8行使用注解，格式 @类名，有些类名里面会有参数，比如此注解的 "a"。

//

(2)、原理

为什么 PHP 普通的注释会有功能？其实很简单，通过 PHP 反射获取类里面所有注解(https://www.php.net/manual/zh/book.reflection.php), 其次通过 PHP 组件(https://github.com/doctrine/annotations) 使其实现特殊功能。

--

--

二、配置

( https://www.swoft.org/docs/2.x/zh-CN/config/index.html )

0、配置组件

Swoft 配置由两部分组成，env 环境配置和 config 应用配置。

env 一般配置一些和环境相关的一些参数，比如运行模式、资源地址。

config 一般用于配置应用级别的配置以及业务级别的配置。

//

(1)、安装

// 安装配置

composer require swoft/config

//

(2)、Git仓库

Github https://github.com/swoft-cloud/swoft-config

//

(3)、参与贡献

欢迎参与贡献，您可以如此。

fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )。

修改代码然后发起 PR。

关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、环境配置

( https://www.swoft.org/docs/2.x/zh-CN/config/env.html )

(1)、配置方式

///1/ 文件配置

项目根目录配置一个名称为 .env 文件，采用 KV 格式配置，此文件配置的数据，可以加载到内存里面，供业务使用。

APP\_DEBUG = 1

SWOFT\_DEBUG = 1

//

///2/ 系统变量

除文件方式配置外，还可以把一些参数配置到系统变量，系统变量的参数也会加载到内存，供业务使用。

//

(2)、如何使用

swoft 提供了函数读取以上两种方式配置的数据。

env(string $key = null, mixed $default = null): mixed

获取一个环境变量的值或所有环境变量参数。

A、返回环境变量 key 的值， 如果环境变量 key 不存在则返回默认值。 如果省略 key 参数，则所有环境变量都将作为关联数组 array 返回。

B、default 默认值，可以是使任何类型，也可以是一个 闭包。

C、返回值默认做了转换。比如配置 true 字符串，返回的转换成一个 bool 类型。

//

(3)、特殊规则

(配置(string) 转换类型结果)

true/false/(true)/(false) bool。

empty string(空字符串)。

null  null。

A\_B 如果是存在的常量，转成对应的值。

--

2、应用配置

( https://www.swoft.org/docs/2.x/zh-CN/config/config.html )

(1)、bean配置

应用配置数据是由一个bean对象管理的，可以在 app/bean.php 文件设置应用配置参数。

return [

    'config'   => [

        'path' => \_\_DIR\_\_ . '/../config',

    ],

];

//

///1/ 可配置项

path 自定配置文件路径

base 主文件名称，默认 base (其他文件的数据都会按文件名为key合并到主文件数据中)。

type 配置文件类型，默认 php 同时也支持 yaml 格式。

parser 配置解析器，默认已经配置 php/yaml 解析器。

env 配置当前环境比如 dev/test/pre/pro。

//

(2)、数据格式

配置目录所有配置文件会解析成一个数组，但是不会递归合并数据，只会合并当前目录文件数据，以它的文件名称为数组 key 进行合并数组。 比如 config 目录配置文件如下。

|-- base.php

|-- data.php

`-- pro

    |-- base.php

    `-- data.php

(只会解析当前目录文件数据，不会递归解析数据。当前使用 env 配置时，环境目录里面的配置信息会覆盖最外层文件名称相同的数据。提醒：配置文件里面可以使用 env()函数读取环境配置)

///1/ config/base.php

return [

    'key' => 'value'

];

///2/ config/data.php

return [

    'dkey' => [

        'dvalue'

    ],

    'key' => 'value'

];

///3/ config/pro/base.php

return [

    'key' => 'valuePro'

];

///4/ config/pro/data.php

return [

    'dkey' => [

        'dvalue'

    ],

    'key' => 'valuePro'

];

//

如上配置文件，当不配置 config 的 env 参数，合并的数据格式如下。

return [

    'key' => 'value',

    'data' => [

        'dkey' => [

            'dvalue'

        ],

        'key' => 'value'

    ]

];

//

/当配置 config 对象的 'env' => 'pro' 参数，合并的数据格式如下。

return [

    'key' => 'valuePro',

    'data' => [

        'dkey' => [

            'dvalue'

        ],

        'key' => 'valuePro'

    ]

];

//

(3)、获取配置

框架提供全局函数、注解、config 对象多种方式，使用应用配置数据。

///1/ 函数

全局函数使用 config()。

config(string $key = null, mixed $default = null):mixed

key 配置参数 key，子数组可以使用 . 分割，比如上面的例子 data.dkey 可以获取到 ["dvalue"], 当key=null 获取所有配置参数。

default 默认参数，如果 key 参数不存在，返回默认值，默认值可以是任意类型。

//

///2/ 注解

通过容器使用注解的方式，注入配置到属性值。

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Config\Annotation\Mapping\Config;

/\*\*

\* @Bean()

\*/

class Demo

{

    /\*\*

    \* @Config("data.dkey")

    \*/

    private $dvalue = [];

    // ...

}

此例子和上面功能一样，都是读取相同的数据，两种不同的方式。使用注解，一定要保证类是一个bean对象(通过其它注解注入到容器)。

//

///3/ 对象

如果上面两种方式还不能满足你的业务需求，你可以从容器里面获取配置对象，里面自带很多方式操作配置数据。

$config = \Swoft::getBean('config');

//

// config 对象常用方法

get(string $key, $default = null) 获取参数。

offsetGet($key) 获取参数。

--

--

三、容器

( https://www.swoft.org/docs/2.x/zh-CN/bean/index.html )

--

0、容器

(1)、前言

Swoft 基于 PSR-11 规范设计了自己容器，并基于 注解 增强了它的功能。 容器是 Swoft 最重要的一个设计，可以说是 Swoft 的核心精髓，也是 Swoft 各种模块的实现基础，本章节将会对容器的相关基础知识做一个介绍，以便新手朋友们更好的理解容器。

//

(2)、什么是容器

我们先看一下比较官方的定义。控制反转（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体，将其所依赖的对象的引用传递(注入)给它。--《维基百科》

这里并没有说到容器，而只是说到了一个叫 IOC 的设计原则，还有一个叫 DI 依赖注入的名词，好像并没有容器的什么事情，这是为什么呢？要搞清楚这个问题我们首先还是需要对一些专业术语进行一个解释。

//

(3)、DIP - 依赖倒置原则

依赖倒置(Dependence Inversion Principle,简称DIP，是一种抽象的软件设计原则，它主要是告诉我们一些规范我们先看下官方对于这个原则的定义。高层模块不应依赖于低层模块，两者应该依赖于抽象。抽象不应该依赖于实现，实现应该依赖于抽象)

具体是什么意思呢我们可以通过一个例子来说明下这个原则，大家应该都对电脑上的 USB 插口很熟悉，通过它我们可以扩展各种的外设能力，例如 U盘、游戏鼠标等等，只要接口一致，插上设备就能正常工作。

在这个例子中，我们的电脑就相当于是高层模块，而 U盘、鼠标等就相当于是底层模块。我们的电脑定义了一个插口（接口），可以供其他的设备插入使用，但是我们的电脑并不依赖于具体要插入的设备，它只是定义好了一个接口规范，只要是符合这个接口规范的设备都可以插入到这台电脑上来使用。那么对应到软件开发上来看指的就是 依赖倒置原则转换了依赖关系，要求高层的模块不应该依赖于底层模块的实现，底层的模块要依赖于高层模块定义的接口。

上面的例子只是说清楚了 依赖倒置原则(DIP) 的规范定义，下面我们再通过一个软件开发的场景例子看看为何需要设计这个原则。

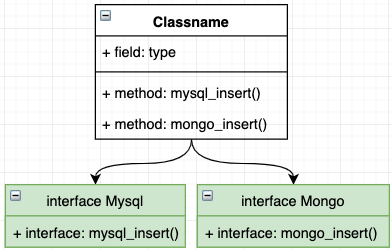
//

///1/ 场景：比如我们有一个业务需要对数据做入库操作，目标可能是 MySql、MongoDb 等。

//

实现一：无依赖倒置，即底层定义接口，高层模块实现。

(图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/bean/NODIPUML.png)

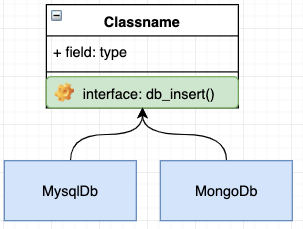


如图就是在不实用依赖倒置的情况下，我们要完成此场景的功能非常的麻烦，我们的业务类（高层模块）要实现所有的 DB类（底层模块） 接口，如果再有新的 DB类（底层模块）介入，则又需要去修改业务类（高层模块）来实现新的 DB类（底层模块）这就破坏了类的开放封闭原则。

//

实现二：使用依赖倒置，即高层定义接口，底层模块实现。

(图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/bean/DIPUML.png)



如图，我们使用依赖倒置后，我们的业务类（高层模块）将不再依赖 DB类（底层模块）而是由 DB类（底层模块）负责实现业务类（高层模块）的接口，这样即使由新的底层模块加入到业务中时我们也不需要修改业务模块。

//

///2/ 使用 DIP 的好处

A、可以通过抽象使各个类或模块的实现彼此独立，不互相影响，实现模块间的松耦合（也是本质）；

B、可以规避一些非技术因素引起的问题（如项目大时，需求变化的概率也越大，通过采用依赖倒置原则设计的接口或抽象类对实现类进行约束，可以减少需求变化引起的工作量剧增情况。同时，发生人员变动，只要文档完善，也可让维护人员轻松地扩展和维护）；

C、可以促进并行开发（如，两个类之间有依赖关系，只要制定出两者之间的接口（或抽象类）就可以独立开发了，而且项目之间的单元测试也可以独立地运行，而TDD开发模式更是DIP的最高级应用（特别适合项目人员整体水平较低时使用））。

//

(4)、IOC - 控制反转

通过上节的介绍我们清楚了DIP-依赖倒置原则，但是它只是一种 软件设计原则 它仅仅是为我们提供一个标准，以便我们可以遵循，避免不良的设计，但是它并不会告诉我们这些标准如何实现。

那么这里就要向大家介绍 IOC 了，它是一种 软件设计模式 即它为我们如何实现 DIP 提供了详细的解决方案。它的定义是：为相互依赖的组件提供抽象，将依赖的获取交给第三方来控制，即依赖对象不在被依赖的模块中获取。

在DIP - 依赖倒置原则 小节中我们列举了USB插口的例子，那么 IOC 的意思通俗来理解就是，我们的电脑并没有插入具体设备的功能，而是由人为（第三方）插入来控制的。下面我们就以之前在 DIP 中的软件开发场景的例子继续深入来介绍 IOC 。

比如说我们的业务是一个订单入库的操作，在最初我们只是将数据存入到 Mysql 中，通常我们会封装一个 MysqlDb 的类用于数据库的操作。

/\*\*

 \* Class MysqlDb

 \*

 \* @since 2.0

 \*/

class MysqlDb

{

    public function insert()

    {

        //TODO::插入一些数据

    }

}

//

//// 紧接着我们看一下我们的业务类

/\*\*

\* Class Order

\*

\* @since 2.0

\*/

class Order

{

    public function add()

    {

        //TODO::订单业务

        $db = new MysqlDb();//建立依赖

        $db->insert();//执行入库操作

    }

}

//

至此我们看似是完成了我们的需求，那这时如果让你改用 MongoDb 怎么办？那我们要先去写一个 MongoDb 的操作类，然后再去我们的 Order 类中修改 DB 的依赖，例如： 定义 Mongo Db 类。

/\*\*

 \* Class MongoDb

 \*

 \* @since 2.0

 \*/

class MongoDb{

    public function insert()

    {

        //TODO::插入一些数据

    }

}

//

然后我们继续修改我们的业务类。

/\*\*

\* Class Order

\*

\* @since 2.0

\*/

class Order

{

    public function add()

    {

        //TODO::订单业务

//        $db = new MysqlDb();//将MysqlDb更改为MongoDb

        $db = new MongoDb();//建立依赖

        $db->insert();//执行入库操作

    }

}

显然这是一个非常糟糕的设计，组件之间还是高度耦合的，同时也破坏了开放封闭原则，也违背了DIP原则。高层的业务模块不应该维护依赖关系，两者应该将依赖抽象出来。那么之前说过 IOC 的出现好像就是为了解决这个问题的，那么它提供了什么方法呢？

IOC 是一个很大的概念，基于这个模式可以有很多种实现方式，但是其主流的有两种：依赖查找(Dependency Lookup 简称DL),依赖注入(Dependency Injection 简称 DI) ,Swoft 使用的是 依赖注入(DI) 技术。我们也主要对 依赖注入DI 进行一个说明介绍，对 依赖查找（DL）感兴趣的朋友可自行 Google。

//

(5)、DI - 依赖注入

DI 是实现 IOC 的一种重要方式，如之前的例子所述，我们将依赖关系在业务中进行创建和绑定是非常糟糕的做法，依赖注入DI 就是解决这种问题，它提供一种实现方式，将需要依赖的底层模块（MysqlDb，MongoDb等）的对象的引用传递给被依赖对象（业务模块）去使用，那么它是如何实现的呢？

DI 主要是由两种方式来实现依赖注入：构造函数注入、属性注入。

构造函数注入顾名思义，它就是通过对象的构造函数将所需的依赖模块传递给对象使用。下面我们来看一下例子。

由于DIP 的原则，我们不应该在模块内部来创建依赖关系即高层模块不应该依赖于底层模块，两者应该依赖于抽象，所以我们先来定义一个接口。

/\*\*

 \* Interface DbDrive

 \*

 \* @since 2.0

 \*/

interface DbDrive

{

    public function insert();

}

//

//// 然后我们将我们的数据库类都实现这个接口。

/\*\*

 \* Class MysqlDb

 \*

 \* @since 2.0

 \*/

class MysqlDb implements DbDrive

{

    public function insert()

    {

        //TODO::插入一些数据

    }

}

/\*\*

 \* Class MongoDb

 \*

 \* @since 2.0

 \*/

class MongoDb implements DbDrive

{

    public function insert()

    {

        //TODO::插入一些数据

    }

}

//

//// 然后改写一下我们的业务类

/\*\*

 \* Class Order

 \*

 \* @since 2.0

 \*/

class Order

{

    /\*\*

     \* @var DbDrive

     \*/

    private $db;

    /\*\*

     \* Order constructor.

     \*

     \* @param DbDrive $driver

     \*/

    public function \_\_construct(DbDrive $driver)

    {

        $this->db = $driver;

    }

    public function add()

    {

        //TODO::订单业务

        $this->db->insert();//执行入库操作

    }

}

//

至此我们就已经完成了构造函数注入的实现方式，这样我们就不需要在业务中（高层模块）关心我该依赖于谁来做那些事情，而是通过第三方（还记得之前举例子的人插u盘的行为）来完成依赖关系的创建，体现为代码就是这样。

$db = new MysqlDb();//创建一个依赖，这就好比是一个u盘

$order = new Order($db);//将需要依赖的对象通过构造函数传递进去，这就好比插入u盘

$order->add();//正常的去调用业务。

这样，我们就将我们的依赖关系从内部转移到了外部，其实这也就是IOC的核心思想，就是高层做接口，比如我们这时候需要更换Db驱动为 redis，则我们只需要编写 redis 类并实现 DbDrive 接口即可然后在业务调用的地方直接将 redis 类通过 构造函数注入 或者 属性注入 的方式注入需要的依赖就可以了，而无需修改我们的 业务类 。属性注入 的方式也是类似的操作只是注入的方式不同，是通过属性注入的这里就不做过多赘述了。

通过以上的介绍我们已经其实已经实现了 DIP,IOC,DI，也都清楚了它们都是什么，那么它们和容器的关系是什么呢？容器又是谁的容器？做了什么呢？下面我们对这些问题进行一些讲解。

//

(6)、IOC Container - IOC容器

容器又叫 IOC 容器，通过之前章节的案例我们通过 DI 实现了 IOC 控制反转，但是我们发现我们要手动的去创建依赖对象，然后再传递给高层模块去使用，显然这样的方式还是有缺陷的，并且效率很低，甚至会出现难以掌控的问题出现。假设我们的业务有十几上百个依赖，并且还存在依赖嵌套等问题，实际工作中这种情况便会很难处理，我们用伪代码来描述下这种情况（试着从最后一行向上阅读）。

$validator = new Validator();//最后一层的依赖。

$check = new Check($validator);//我们的业务检查类，同时它又依赖于一个验证器。

$db = new Mysql();//Db类。

$user = new User($db,$check);//我们的用户类业务，同时它又依赖于一个db类,和一个检查类业务。

$order = new Order($user);//我们的订单业务，它依赖于一个用户类业务。

此时就暴露出来了很多问题，包括但不限于之前的问题，例如还有对象生命周期等问题。这时出现了一个新的技术那就是 IOC 容器，就是用来解决上述的问题，他的主要功能就是如下如此。

//

//// IOC容器主要功能

自动的管理依赖关系，避免手工管理的缺陷。

在需要使用依赖的时候自动的为我们注入所需依赖。

管理对象的生命周期。

//

为了更好的理解容器，我们来实现一个简单的通过 构造函数注入 的容器，（注意 这只是为了科普教学，Swoft已经为大家准备了更加完善强大且易用的 IOC 容器）。

<?php

/\*\*

 \* Class Container

 \*/

class Container

{

    /\*\*

     \* 容器内所管理的所有实例

     \* @var array

     \*/

    protected $instances = [];

    /\*\*

     \* @param $class

     \* @param null $concrete

     \*/

    public function set($class, $concrete = null)

    {

        if ($concrete === null) {

            $concrete = $class;

        }

        $this->instances[$class] = $concrete;

    }

    /\*\*

     \* 获取目标实例

     \*

     \* @param $class

     \* @param array $param

     \*

     \* @return mixed|null|object

     \* @throws Exception

     \*/

    public function get($class, ...$param)

    {

        // 如果容器中不存在则注册到容器

        if (!isset($this->instances[$class])) {

            $this->set($class);

        }

        //解决依赖并返回实例

        return $this->resolve($this->instances[$class], $param);

    }

    /\*\*

     \* 解决依赖

     \*

     \* @param $class

     \* @param $param

     \*

     \* @return mixed|object

     \* @throws ReflectionException

     \* @throws Exception

     \*/

    public function resolve($class, $param)

    {

        if ($class instanceof Closure) {

            return $class($this, $param);

        }

        $reflector = new ReflectionClass($class);

        // 检查类是否可以实例化

        if (!$reflector->isInstantiable()) {

            throw new Exception("{$class} 不能被实例化");

        }

        // 通过反射获取到目标类的构造函数

        $constructor = $reflector->getConstructor();

        if (is\_null($constructor)) {

            // 如果目标没有构造函数则直接返回实例化对象

            return $reflector->newInstance();

        }

        // 获取构造函数参数

        $parameters = $constructor->getParameters();

        //获取到构造函数中的依赖

        $dependencies = $this->getDependencies($parameters);

        // 解决掉所有依赖问题并返回实例

        return $reflector->newInstanceArgs($dependencies);

    }

    /\*\*

     \* 解决依赖关系

     \*

     \* @param $parameters

     \*

     \* @return array

     \* @throws Exception

     \*/

    public function getDependencies($parameters)

    {

        $dependencies = [];

        foreach ($parameters as $parameter) {

            $dependency = $parameter->getClass();

            if ($dependency === null) {

                // 检查是否有默认值

                if ($parameter->isDefaultValueAvailable()) {

                    // 获取参数默认值

                    $dependencies[] = $parameter->getDefaultValue();

                } else {

                    throw new Exception("无法解析依赖关系 {$parameter->name}");

                }

            } else {

                // 重新调用get() 方法获取需要依赖的类到容器中。

                $dependencies[] = $this->get($dependency->name);

            }

        }

        return $dependencies;

    }

}

class MysqlDb

{

    public function insert()

    {

        echo 'mysql';

    }

}

//

class Order

{

    private $db;

    public function \_\_construct(MysqlDb $db)

    {

        $this->db = $db;

    }

    public function add()

    {

        $this->db->insert();

    }

}

//

$container = new Container();//使用容器。

$order = $container->get('Order');//通过容器拿到我们的Order类。

$order->add();//正常的使用业务。

//

我们主要是利用反射类来完成容器的自动注入，换句话说容器其实就像是一个工厂模式，使用容器就和使用工厂类似的，它会帮我们解决好依赖关系然后再返回给我们对象示例，上方只是个简单的演示，还有许多地方并没有考虑到，比如循环机制的处理，对象的缓存，生命周期管理等。

不过 Swoft 框架已经为我们提供了非常完整易用的 IOC 容器，我们会在后续的章节详细介绍如何使用它。

//

(7)、什么是 Bean

介绍到了最后还有一个小概念，那就是 Bean 有了之前的知识铺垫，我们能很快的描述清楚什么是 Bean。

我们先看下定义：Bean 是由 IOC 容器管理的实例。也就是说其实 Bean 就是一个类的对象实例，只不过它是由 IOC 容器 实例化、组装和管理的对象。

IOC容器可以看成是一个Beans关系的集合 ,我们的应用程序就是由很多个Bean组成的。

BeanFactory提供了一种先进的配置机制来管理任何种类的Bean。

Bean的定义要有BeanDefinition描述：当配置文件/注解被解析后就会在内部转化成一个BeanDefinition对象。以后的操作都是对这个对象完成的。

//

(8)、哪些是Bean

Bean 不仅仅等于 `@Bean`，虽然大多数情况下指的是同一个东西。如下的，所有类注解标记的class，放到容器里都可以称之为Bean对象。

//

//// 类注解

@Bean 最常用的bean注解

@Listener

@Controller

@Command

@WsModule

@WsController

等等...

//

--

1、Bean

(https://www.swoft.org/docs/2.x/zh-CN/bean/bean.html)

Swoft 中一个 Bean 就是一个类的一个对象实例。 容器就是一个巨大的工厂，用于存放和管理 Bean 生命周期。

//

(1)、创建 Bean

直接定义注解 [@Bean](#@Bean()) 的方式。

通过在bean.php中配置。

在AutoLoader.php 中定义。

//

///1/ @Bean()

<?php declare(strict\_types=1);

namespace SwoftTest\Bean\Testing\Definition;

use Swoft\Bean\Annotation\Mapping\Bean;

/\*\*

 \* Class PrototypeClass

 \*

 \* @since 2.0

 \*

 \* @Bean(name="prototype", scope=Bean::PROTOTYPE, alias="pro")

 \*/

class PrototypeClass

{

    // ....

}

?>

scope：Bean::SINGLETON 单例Bean( https://www.swoft.org/docs/2.x/zh-CN/bean/singleton.html )；Bean::PROTOTYPE 原型Bean( https://www.swoft.org/docs/2.x/zh-CN/bean/prototype.html )；Bean::REQUEST 请求Bean( https://www.swoft.org/docs/2.x/zh-CN/bean/request.html )。

name：为 bean 指定一个名称，有时候你可能只需要配置一个 name 可以这样写@Bean("xxx")这样默认是参数的 name。

alias：为 bean 指定一个别名, 你可以通过别名获取它。

@Bean 定义的在初始化的时候如果有构造函数需要注入参数 请给默认值，然后通过配置( https://www.swoft.org/docs/2.x/zh-CN/bean/bean.html#Config )的覆盖构造函数所需参数。scope 注入 Bean 类型，默认单例。

//

///2/在 bean.php配置文件中,创建 Bean

例如下面这个例子,下面我要利用配置文件将它 加入到 Bean 中。

<?php declare(strict\_types=1);

namespace SwoftTest\Bean\Testing\Definition;

/\*\*

 \* TestBean

 \*

 \* @since 2.0

 \*/

class TestBean

{

    /\*\*

     \* @var SingletonClass

     \*/

    private $singleton;

     /\*\*

     \* @var \JWT

     \*/

    private $jwt;

    /\*\*

     \* TestBean constructor.

     \*

     \* @param SingletonClass $singleton

     \*/

    public function \_\_construct(SingletonClass $singleton = null)

    {

        $this->singleton = $singleton;

    }

}

?>

// 配置方式 指定 class 的方式

copy

 'testBean'     => [

     'class'    => TestBean::class,

     [\bean('singleton')],

     'jwt'      => \bean('jwt'),

     '\_\_option' => [

         'scope' => Bean::PROTOTYPE,

         'alias' => 'testBean-alias'

     ]

 ],

testBean 这个 key 你可以理解是一个Bean的名称。

class 参数的指定 Bean 使用哪一个类。

示例中的数组[\bean('singleton')], 是 TestBean类构造函数所需参数,不推荐使用构造注入，请使用@Inject。

\_\_option 这个swoft 自带的你可以指定：scope 指定 Bean 是用哪个级别,如果没有默认单例模式；alias 指定 Bean 的别名。

Swoft 的 类构造方法, 不会自动依赖注入。

//

//// 当然你也可以注入自己定义的属性

jwt 这个是上面TestBean类定义的属性，底层会通过反射注入config中的参数。

通过 配置文件配置的 Bean 优先级最高因为它是最后执行的，如果配置的已经是一个 Bean ，config的 配置的将会覆盖它。

//

///3/在AutoLoader.php 中定义

<?php declare(strict\_types=1);

namespace App;

use Swoft\Db\Database;

use Swoft\SwoftComponent;

/\*\*

 \* Class AutoLoader

 \*

 \* @since 2.0

 \*/

class AutoLoader extends SwoftComponent

{

    /\*\*

     \* @return array

     \*/

    public function getPrefixDirs(): array

    {

        return [

            \_\_NAMESPACE\_\_ => \_\_DIR\_\_,

        ];

    }

    /\*\*

     \* @return array

     \*/

    public function metadata(): array

    {

        return [];

    }

    /\*\*

     \* 使用 beans 方法 和在 bean.php 中配置的一样的使用方式,它的优先级在bean.php定义的 bean 之下

     \*

     \* @return array

     \*/

    public function beans(): array

    {

        return [

            'db' => [

                'class'    => Database::class,

                'dsn'      => 'mysql:dbname=test;host=172.17.0.4',

                'username' => 'root',

                'password' => 'swoft123456',

            ],

        ];

    }

}

?>

//

(2)、Bean 初始化

每一个 Bean初始化的时候会自动检查init()这个方法是否存在。会在Bean初始化完之后调用，你可以在该方法中进行一些初始化操作， 作用类似 golang 中的init方法。

//

(3)、获取 Bean

///1/ @Inject

score为 Bean::SINGLETON 级别的 bean 可以通过 @Inject 属性注入，底层会通过反射自动注入属性中。Bean::PROTOTYPE, Bean::REQUEST bean不能使用 @Inject 注入。

/\*\*

 \* @Inject("config")

 \*

 \* @var Config

 \*/

private $config;

name 定义属性注入的bean名称。如果name为空，默认为@var 定义的类型。 这个name 可以是一个完整的类名，也可以是bean别名/bean名称。如果 @Inject 没有指定 Bean 名称, 会自动根据 @var 的类型查找 Bean。

tips： 如果要使用 @Inject 属性注入必须有 类注解才可以，不然不会被解析。

//

///2/ BeanFactory

BeanFactory提供了一种先进的配置机制来管理任何种类的bean。获取score为Bean::SINGLETON，Bean::PROTOTYPE。可以使用下面，以获取 wsRouter 为例。

/\*\* @var Router $router \*/

$router = Swoft::getBean('wsRouter');

$router = BeanFactory::getBean('wsRouter');

$router = Container::$instance->get('wsRouter')

$router = BeanFactory::getContainer()->get('wsRouter')

上述这几种方式都可以获取Bean::SINGLETON或Bean::PROTOTYPE类型的bean，你也可以指定类型获取。

//

//// 获取单例类型的bean

注意BeanFactory::getBean只能获取到 框架启动加载的 bean，不能获取到 scope 为 request 的bean。request bean( https://www.swoft.org/docs/2.x/zh-CN/bean/request.html ) 的类型是 2.x 新加的。获取 score为 request类型的 你可以这样获取。

$requestBean = BeanFactory::getRequestBean(RequestClass::class, (string)Co::tid());

一般 request 类型的 bean 通常是与顶级协程ID绑定在一起的。

//

//// 判断当前环境是否存在 某个 bean

$exist = BeanFactory::hasBean($name);

//

//// 判断是否是一个单例的 bean

$isSingleton = BeanFactory::isSingleton('name');

--

2、单例 singleton

( https://www.swoft.org/docs/2.x/zh-CN/bean/singleton.html )

单例Bean全局只有一个共享的实例，所有将单例Bean作为依赖的情况下，容器返回将是同一个实例。

换言之，当开发者定义一个Bean的作用域为单例时，Swoft IOC容器只会根据Bean定义来创建该Bean的唯一实例。这些唯一的实例会缓存到容器中，后续针对单例Bean的请求和引用，都会从这个缓存中拿到这个唯一的实例。

(1)、简介

singleton的bean 会在框架启动时被初始化，并且只会初始化一次。 对singleton的 bean 进行写入操作是不安全的。如果同时读写singleton的 bean，会造成上下文切换导致bean数据不一致，这种往往是业务交叉造成的。

//

(2)、特别注意

singleton 一般用于只读， 不要把它当做共享内存的方式。例如 swoft 的 config bean，就是一个全局单例的。它只用于读取，在运行过程中不会发生改变。系统把config 写操作都抛出了异常。

//

(3)、定义

<?php declare(strict\_types=1);

namespace Swoft\Config;

use Swoft\Bean\Annotation\Mapping\Bean;

/\*\*

 \* Class Config

 \*

 \* @Bean("config")

 \*

 \* @since 2.0

 \*/

class Config

{

}

默认是 score 是全局单例的，创建单例的bean，你可以不用指定score。

//

(4)、销毁

singleton 类型的 bean 只会在 主进程关闭 才会被销毁。

--

3、Prototype 原型实例

( https://www.swoft.org/docs/2.x/zh-CN/bean/prototype.html )

prototype 用的是原型模式, 它会被框架启动时会被自动初始化。

//

(1)、原型模式

获取 scope为prototype类型的bean每次都是克隆初始化好的bean。clone 一个对象 比 重新new一个对象更快, 因为它是拷贝操作。在 swoft 中 DB 的collection 就是用的prototype类型。

//

(2)、如何使用

你可以定义一个 new 方法，替代new关键字。比如Db使用的Collection Prototype：实体都是 prototype，类型的bean，所有实体都可以使用new方法。

<?php declare(strict\_types=1);

namespace Swoft\Test;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Bean\Concern\PrototypeTrait;

/\*\*

 \* Class TestCollection

 \*

 \* @Bean(scope=Bean::PROTOTYPE)

 \*

 \* @since 2.0

 \*/

class TestCollection

{

    use PrototypeTrait;

    /\*\*

     \* @var array

     \*/

    private $items;

    /\*\*

     \* Create a new collection.

     \*

     \* @param array $items

     \*

     \* @return static

     \*/

    public static function new(array $items = []): self

    {

        $self        = self::\_\_instance();

        $self->items = $items;

        return $self;

    }

}

?>

需要引入PrototypeTrait，在PrototypeTrait 中实现了 \_\_instance()方法，该返回的就是一个 clone 的自身对象，你只需对应更新属性值 即可获取一个全新的对象。

--

4、请求实例 (Request Bean)

( https://www.swoft.org/docs/2.x/zh-CN/bean/request.html )

scope 为 request 类型的 bean， 框架初始化的时候并不会初始化，而是在 onRequest 事件触发后 采用懒加载方式，在框架初始化的时候不会初始化。只有真正调用的时候才会被初始化，在当前请求中保持单例，请求结束后会被自动销毁。作用域始终在于一个请求当中。

--

(1)、使用

只能通过获取 BeanFactory::getRequestBean 获取。

$obj = BeanFactory::getRequestBean($name, (string) $tid);

name requestBean 的名称/别名/类名。

tid 通常是与是顶级协程ID绑定。获取顶级协程ID，Co::tid()获取当前协程环境 顶级协程ID。

如果你需要获取当前请求加载了那些 request bean 。可以使用如下方式。

$pool = BeanFactory::getContainer()->getRequestPool()。

//

(2)、销毁

在所有协程执行完毕后，在SwoftEvent::COROUTINE\_COMPLETE事件中， 会自动销毁与顶级协程ID绑定的request bean。这时 request bean的生命周期也就结束了。

--

5、接口注入

( https://www.swoft.org/docs/2.x/zh-CN/bean/interface.html )

在实际业务场景中，很多情况需要面向接口编程，需要根据接口类名，注入接口实际实现的对象。面向接口编程可以很方便的解耦业务，如下以发送短信为例，2.0.3 开始支持。

--

(1)、注解注入

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Bean\Annotation\Mapping\Primary;

use Swoft\Bean\BeanFactory;

interface SmsInterface

{

    public function send(string $content): bool;

}

/\*\*

 \* Class AliyunSms

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \* @Primary()

 \*/

class AliyunSms implements SmsInterface

{

    /\*\*

     \* @param string $content

     \*

     \* @return bool

     \*/

    public function send(string $content): bool

    {

        return true;

    }

}

/\*\*

 \* Class QcloudSms

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class QcloudSms implements SmsInterface

{

    /\*\*

     \* @param string $content

     \*

     \* @return bool

     \*/

    public function send(string $content): bool

    {

        return true;

    }

}

/\*\*

 \* Class Sms

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class Sms implements SmsInterface

{

    /\*\*

     \* @Inject()

     \*

     \* @var SmsInterface

     \*/

    private $smsInterface;

    /\*\*

     \* @param string $content

     \*

     \* @return bool

     \*/

    public function send(string $content): bool

    {

        return $this->smsInterface->send($content);

    }

}

/\* @var SmsInterface $sms\*/

$sms = BeanFactory::getBean(Sms::class);

// Send Aliyun sms

$sms->send('sms content');

?>

当前是以发送阿里云短信为例，如果业务后续变化，需要使用 Gcloud 送短信，只需移除 AliyunSms 的 @Primary()注解，把此注解加到 Qcloud 类上面，完全解耦业务代码，无任何修改。@Inject() 注入 bean 名称为空，会优先注入 @Primary() 注解标记的实例对象，如果不为空比如 @Inject(QcloudSms::class) 就会注入指定的实例对象。

//

(2)、@Primary

如果一个接口有多个实现，此标记就是制定接口注入时使用的对象，且同一个接口的多个实现类中只能有一个此注解标记，不然启动会提示错误。

//

(3)、配置注入

bean 配置 (app\bean.php) 也是支持接口注入的，如上例子把发送短信的 Sms 修改下不用注解，其它不变，代码如下方式。

/\*\*

 \* Class Sms

 \*

 \* @since 2.0

 \*/

class Sms implements SmsInterface

{

    /\*\*

     \* @var SmsInterface

     \*/

    private $smsInterface;

    /\*\*

     \* @param string $content

     \*

     \* @return bool

     \*/

    public function send(string $content): bool

    {

        return $this->smsInterface->send($content);

    }

}

//

//// app\bean.php 文件中配置bean

//

[

   Sms::class => [

        'smsInterface' => bean(SmsInterface::class)

   ]

]

//

//// 使用代码保存不变，其实现的功能与注解方式是一样的

/\* @var SmsInterface $sms\*/

$sms = BeanFactory::getBean(Sms::class);

//

// Send Aliyun sms

$sms->send('sms content');

//

--

四、事件

( https://www.swoft.org/docs/2.x/zh-CN/event/index.html )

//

0、事件管理

( https://www.swoft.org/docs/2.x/zh-CN/event/index.html )

Swoft 2 事件进行了更加清晰和严谨的规划。提供了基本的事件注册与触发管理。

//

(1)、功能说明

implement the Psr 14 - Event dispatcher。

支持通过注解快速的注册事件，事件组。

支持设置事件优先级。

支持对通配符事件的监听。

作为Swoft的核心组件，事件管理会自动启用

//

(2)、安装

composer require swoft/event

//

(3)、Git仓库

Github https://github.com/swoft-cloud/swoft-event

//

(4)、参与贡献

欢迎参与贡献，您可以：

fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )。

修改代码然后发起 PR。

关于发起PR的注意事项( https://github.com/swoft-cloud/swoft-component )。

//

(5)、swoft里的事件

Swoft 2 事件进行了更加清晰和严谨的规划。提供了丰富的事件，以便于开发者使用。在swoft我们将事件分为三大类：swoole server的回调事件；swoft server的事件，基于swoole的回调处理，扩展了一些可用事件以增强自定义性；应用级别内的自定义事件管理和使用。

//

(6)、相关介绍

一些关于自定义事件的拓展介绍说明。

///1/ 事件分组

除了一些特殊的事件外，在一个应用中，大多数事件是有关联的，此时我们就可以对事件进行分组，方便识别和管理使用。

//

//a/ 事件分组 推荐将相关的事件，在名称设计上进行分组

swoft.server.\*

swoft.process.\*

swoft.pool.\*

swoft.http.request.before

swoft.http.request.after

swoft.db.query.start

swoft.db.query.after

swoft.redis.start

swoft.redis.after

swoft.ws.start

swoft.ws.after

swoft.tcp.start

swoft.tcp.after

swoft.udp.start

swoft.udp.after

//

///2/ 事件通配符 \*

支持使用事件通配符 \* 对一组相关的事件进行监听, 分两种。\*全局的事件通配符。直接对 \* 添加监听器(@Listener("\*")), 此时所有触发的事件都会被此监听器接收到。

{prefix}.\* 指定分组事件的监听。例如 @Listener("swoft.db.\*"), 此时所有触发的以 swoft.db. 为前缀的事件(eg swoft.db.query swoft.db.connect)都会被此监听器接收到。

当然，你在事件到达监听器前停止了本次事件的传播$event->stopPropagation(true)，就不会被后面的监听器接收到了。

///3/ 更多介绍

更多关于自定义事件的理解参考 https://github.com/inhere/php-event-manager/blob/master/README.md。

//

1、监听与触发 (事件注册)

( https://www.swoft.org/docs/2.x/zh-CN/event/usage.html )

swoft提供了简便的事件管理和使用

//

(1)、注解

///1/ Listener

事件监听器类注解tag @Listener

注解类： Swoft\Event\Annotation\Mapping\Listener；

作用范围： CLASS；

拥有属性：event string 要监听的事件名称；priority int 此监听器的优先级，值越大越先被调用。

注意你的类必须实现接口: Swoft\Event\EventHandlerInterface。

//

///2/ Subscriber

事件监听器类注解tag @Subscriber，与 @Listener 不同的是，允许在同一个类里处理多个事件。

注解类： Swoft\Event\Annotation\Mapping\Subscriber

作用范围： CLASS

注意你的类必须实现接口: Swoft\Event\EventSubscriberInterface。

//

(2)、示例

///1/ Listener示例

<?php declare(strict\_types=1);

namespace SwoftTest\Event\Testing;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

/\*\*

 \* Class TestHandler

 \* @Listener("test.evt")

 \*/

class TestHandler implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        $pos = \_\_METHOD\_\_;

        echo "handle the event '{$event->getName()}' on the: $pos\n";

    }

}

//

///2/ Subscriber示例

<?php declare(strict\_types=1);

namespace SwoftTest\Event\Testing;

use Swoft\Event\Annotation\Mapping\Subscriber;

use Swoft\Event\EventInterface;

use Swoft\Event\EventSubscriberInterface;

use Swoft\Event\Listener\ListenerPriority;

/\*\*

 \* Class TestSubscriber

 \* @Subscriber()

 \*/

class TestSubscriber implements EventSubscriberInterface

{

    public const EVENT\_ONE = 'test.event1';

    public const EVENT\_TWO = 'test.event2';

    /\*\*

     \* Configure events and corresponding processing methods (you can configure the priority)

     \* @return array

     \* [

     \*  'event name' => 'handler method'

     \*  'event name' => ['handler method', priority]

     \* ]

     \*/

    public static function getSubscribedEvents(): array

    {

        return [

            self::EVENT\_ONE => 'handleEvent1',

            self::EVENT\_TWO => ['handleEvent2', ListenerPriority::HIGH],

        ];

    }

    public function handleEvent1(EventInterface $evt): void

    {

        $evt->setParams(['msg' => 'handle the event: test.event1 position: TestSubscriber.handleEvent1()']);

    }

    public function handleEvent2(EventInterface $evt): void

    {

        $evt->setParams(['msg' => 'handle the event: test.event2 position: TestSubscriber.handleEvent2()']);

    }

}

?>

//

(3)、触发事件

事件名称管理推荐放置在一个单独类的常量里面，方便管理和维护。

///1/ 方式一

使用简便。但是多个参数按顺序放入，因此获取时需要根据索引获取。

Swoft::trigger('event name', mixd $target, $args...);

\Swoft::trigger('event name', 'target', $arg0, $arg1);

//

//// 获取事件参数

$target = $event->getTarget();

$arg0 = $event->getParam(0);

$arg1 = $event->getParam(1);

//

///2/ 方式二

使用稍微麻烦一点。但是多个参数按k-v放入，获取时可以根据key获取。

Swoft::triggerByArray('event name', mixd $target, array $args);

\Swoft::triggerByArray('event name', 'target', [

    'arg0' => $arg0,

    'arg1' => $arg1

]);

//

//// 获取事件参数

$target = $event->getTarget();

$arg0 = $event->getParam('arg0');

$arg1 = $event->getParam('arg1');

--

2、swoole 事件

( https://www.swoft.org/docs/2.x/zh-CN/event/swoole-events.html )

swoole文档上列出的每个事件，在swoft里面都可以监听，并且可以有多个监听器。

//

(1)、事件常量

swoft 里我们为这些事件名添加了常量，方便统一使用和管理。

<?php declare(strict\_types=1);

namespace Swoft\Server\Swoole;

/\*\*

 \* Class SwooleEvent

 \*

 \* @since 2.0

 \*/

final class SwooleEvent

{

    /\*\*

     \* Start

     \*/

    public const START = 'start';

    /\*\*

     \* Shutdown

     \*/

    public const SHUTDOWN = 'shutdown';

    /\*\*

     \* WorkerStart

     \*/

    public const WORKER\_START = 'workerStart';

    /\*\*

     \* WorkerStop

     \*/

    public const WORKER\_STOP = 'workerStop';

    /\*\*

     \* WorkerError

     \*/

    public const WORKER\_ERROR = 'workerError';

    /\*\*

     \* ManagerStart

     \*/

    public const MANAGER\_START = 'managerStart';

    /\*\*

     \* ManagerStop

     \*/

    public const MANAGER\_STOP = 'managerStop';

    /\*\*

     \* Task

     \*/

    public const TASK = 'task';

    /\*\*

     \* Finish

     \*/

    public const FINISH = 'finish';

    /\*\*

     \* PipeMessage

     \*/

    public const PIPE\_MESSAGE = 'pipeMessage';

    /\*\*

     \* Handshake

     \*/

    public const HANDSHAKE = 'handshake';

    /\*\*

     \* Message

     \*/

    public const MESSAGE = 'message';

    /\*\*

     \* Open

     \*/

    public const OPEN = 'open';

    /\*\*

     \* Request

     \*/

    public const REQUEST = 'request';

    /\*\*

     \* Packet

     \*/

    public const PACKET = 'packet';

    /\*\*

     \* Receive

     \*/

    public const RECEIVE = 'receive';

    /\*\*

     \* Connect

     \*/

    public const CONNECT = 'connect';

    /\*\*

     \* Close

     \*/

    public const CLOSE = 'close';

    /\*\*

     \* BufferFull

     \*/

    public const BUFFER\_FULL = 'bufferFull';

    /\*\*

     \* BufferEmpty

     \*/

    public const BUFFER\_EMPTY = 'bufferEmpty';

}

//

(2)、监听swoole事件

这里我们监听 master 进程启动时的事件，要监听其他事件也是类似写法。

<?php declare(strict\_types=1);

namespace App\Listener;

use ReflectionException;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Log\Helper\CLog;

use Swoft\Server\Swoole\SwooleEvent;

/\*\*

 \* Class MasterStartListener

 \*

 \* @Listener(SwooleEvent::START)

 \*/

class MasterStartListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*

     \* @throws ReflectionException

     \* @throws ContainerException

     \*/

    public function handle(EventInterface $event): void

    {

        CLog::info('master started');

    }

}

?>

--

3、Swoft事件 (内置事件)

( https://www.swoft.org/docs/2.x/zh-CN/event/swoft-events.html )

swoft 内置事件，基于swoole的回调处理，扩展了一些可用server事件和启动运行时的基础以增强自定义性。

//

(1)、Server 事件

基于swoole的回调处理，扩展了一些可用server事件，提供更加精细化的操作空间。

ServerEvent::BEFORE\_SETTING 在调用 swoole server 的 setting() 方法之前。

ServerEvent::BEFORE\_BIND\_EVENT 在调用 swoole server 的 on() 方法绑定swoole回调之前。

ServerEvent::BEFORE\_BIND\_LISTENER 在调用 swoole server 的 listen() 方法添加多端口监听之前。

ServerEvent::AFTER\_ADDED\_LISTENER 每当调用 swoole server 的 listen() 方法成功添加一个端口监听之后(添加多个端口简单，则会调用多次)。

ServerEvent::BEFORE\_START 在调用 swoole server 的 start() 方法启动server之前。

ServerEvent::TASK\_PROCESS\_START 仅当 swoole server 的task进程启动时触发。

ServerEvent::WORK\_PROCESS\_START 仅当 swoole server 的work进程启动时触发。

//

<?php declare(strict\_types=1);

namespace Swoft\Server;

/\*\*

 \* Class ServerEvent

 \*

 \* @since 2.0

 \*/

final class ServerEvent

{

    /\*\*

     \* Before set swoole settings

     \*/

    public const BEFORE\_SETTING = 'swoft.server.before.setting';

    /\*\*

     \* Before bind swoole events

     \*/

    public const BEFORE\_BIND\_EVENT = 'swoft.server.bind.event';

    /\*\*

     \* Before bind listener(s)

     \*/

    public const BEFORE\_BIND\_LISTENER = 'swoft.server.bind.listener.before';

    /\*\*

     \* After each listener is successfully added

     \*/

    public const AFTER\_ADDED\_LISTENER = 'swoft.server.added.listener.after';

    /\*\*

     \* Swoft before start server event

     \*/

    public const BEFORE\_START = 'swoft.server.start.before';

    /\*\*

     \* On task process start event

     \*/

    public const TASK\_PROCESS\_START = 'swoft.process.task.start';

    /\*\*

     \* On work process start event

     \*/

    public const WORK\_PROCESS\_START = 'swoft.process.work.start';

    /\*\*

     \* on user process start event

     \*/

    public const USER\_PROCESS\_START = 'swoft.process.user.start';

}

//

///1/ 使用示例

我们可以在swoole server启动前注册一个自定义进程，这样可以让进程有server托管。

不需要执行start。在Server启动时会自动创建进程，并执行指定的子进程函数。

在shutdown关闭服务器时，会向用户进程发送SIGTERM信号，关闭用户进程。

自定义进程会托管到Manager进程，如果发生致命错误，Manager进程会重新创建一个。

//

<?php declare(strict\_types=1);

namespace App\Listener;

use App\Process\MyProcess;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Server\ServerEvent;

/\*\*

 \* Class AttachMyProcessHandler

 \* @Listener(ServerEvent::BEFORE\_START)

 \*/

class AttachMyProcessHandler implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        $swooleServer = $event->target->getSwooleServer();

        $process = bean(MyProcess::class);

        $swooleServer->addProcess($process->create());

    }

}

//

<?php declare(strict\_types=1);

namespace App\Process;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoole\Process;

/\*\*

 \* Class MyProcess

 \* @Bean()

 \*/

class MyProcess

{

    public function create(): Process

    {

        $process = new Process([$this, 'handle']);

        return $process;

    }

    public function handle(Process $process)

    {

        CLog::info('my-process started');

        // 用户进程实现了广播功能，循环接收管道消息，并发给服务器的所有连接

        while (true) {

            $msg = $process->read();

            foreach($server->connections as $conn) {

                $server->send($conn, $msg);

            }

        }

    }

}

?>

//

(2)、Swoft 基础事件

提供了一些swoft启动初始化完成后，以及一些swoft内部的特殊事件。

//

///1/ SwoftEvent::APP\_INIT\_COMPLETE 当swoft初始化完成(配置已加载，容器已经初始化完成)后触发

<?php declare(strict\_types=1);

namespace Swoft;

/\*\*

 \* Class SwoftEvent

 \* @since 2.0

 \*/

final class SwoftEvent

{

    /\*\*

     \* Swoft init complete

     \*/

    public const APP\_INIT\_COMPLETE  = 'swoft.init.complete';

    /\*\*

     \* Session complete

     \*  - webSocket connection close

     \*/

    public const SESSION\_COMPLETE = 'swoft.session.complete';

    /\*\*

     \* Coroutine complete

     \*/

    public const COROUTINE\_COMPLETE = 'swoft.co.complete';

    /\*\*

     \* Coroutine destroy

     \*/

    public const COROUTINE\_DESTROY = 'swoft.co.destroy';

    /\*\*

     \* Coroutine defer

     \*/

    public const COROUTINE\_DEFER = 'swoft.co.defer';

    /\*\*

     \* Worker shutdown

     \*/

    public const WORKER\_SHUTDOWN = 'swoft.worker.shutdown';

}

?>

--

5、AOP切面编程

( https://www.swoft.org/docs/2.x/zh-CN/aop/index.html )

--

(0-1)、前言

在介绍 AOP 之前我们先来简单的说明一下，AOP 和 OOP 一样都是一种编程思想，AOP 只是 OOP 的补充和延伸，可以更方便的对业务代码进行解耦，提高代码质量，增加代码的可重用性。后面的篇章会结合实际案例，教会大家理解并学会使用 AOP。

//

(0-2)、场景案例

我们就从日志记录入手带领大家理解AOP切面编程的优势。

///1/ 首先我们假设我们有以下需求。

需要对某个接口做调用日志记录，记录接口的调用详情（调用参数，返回结果等）。当接口调用出现异常时，做特殊处理比如记录日志、通知我们的运维小伙伴等。如上，我们定义了一个简单的需求，接下来我们去实现他，我们先不要考虑 AOP ，就先用你所熟悉的 OOP 思想来完成以上的需求。

//

(0-3)、OOP 实现

///1/ 假设我们有以下的一个业务接口

<?php declare(strict\_types=1);

/\*\*

 \* 订单服务类

 \*

 \* @since 2.0

 \*/

class OrderServer

{

    /\*\*

     \* 下单接口

     \*

     \* @param $productName //商品名字

     \* @param $number //购买数量

     \* @param $uid //用户id

     \*

     \* @return array        //订单参数

     \*/

    public function order($productName, $number, $uid): array

    {

        $price = 1000;

        $total = $price \* $number;

        $data = [

            'product\_name' => $productName,

            'price' => $price,

            'number' => $number,

            'total' => $total,

            'order\_id' => uniqid($uid . time() . $total)

        ];

        return $data;

    }

}

?>

//

在传统的 OOP 思想下，我们可能会写一个类继承于我们的 OrderServer 然后重写其 order 方法，然后在使用 OrderServer 类的地方，替换成我们自己定义的类。示例代码如下：首先我们先编写一个继承于 OrderServer 的子类。

//

<?php declare(strict\_types=1);

/\*\*

 \* Class InheritOrderServer

 \*

 \* @since 2.0

 \*/

class InheritOrderServer extends OrderServer

{

    /\*\*

     \* @param $productName

     \* @param $number

     \* @param $uid

     \*

     \* @return array

     \* @throws Exception

     \*/

    public function order($productName, $number, $uid): array

    {

        //TODO::在这里我们记录接口调用时传过来的参数

        try {

            $result = parent::order($productName, $number, $uid);

            //TODO::这里可以处理父类方法返回的结果 然后返回结果进行后续处理

            return $result;

        } catch (Exception $e) {

            //TODO::如果有出错异常，在这里我们处理出错的日志记录等操作之后继续将异常抛出

            throw $e;

        }

    }

}

?>

//

接下来继续，去找到我们原先调用 OrderServer 的地方，将原先的调用修改成我们后来改写的继承类 InheritOrderServer。

<?php declare(strict\_types=1);

  //$order = new OrderServer(); //屏蔽掉原先的旧的业务调用

  $order = new InheritOrderServer();//修改为我们新继承的子类

  $order->order('iPhone XR', 200, 10000);

?>

至此，经过如上的操作，我们貌似也能满足了业务需求完成了日志记录的功能，但是你会发现过程及其繁杂，代码耦合严重，污染了order方法，如果旧的业务中有100处调用，那么你就要去到业务代码里面修改100处代码,如果你一直以OOP 的思想来试着解决这些问题，貌似很难办，因为几乎都是高耦合的侵入式的解决方案。

这时也许你可能会想到使用中间件，拦截器等等一些方案，但是这些其实说白了已经是基于AOP思想的设计模式了。其实 AOP 就是基于 OOP 的延伸和补充，他就是一种思想，一种设计模式。

//

(0-4)、AOP 实现

接下来我们将基于上面的例子，引入 AOP 的概念，告诉大家，什么是AOP (what) ,如何使用AOP (how), 为什么使用AOP (why) 。

//

(1)、什么是AOP

///1/ 定义

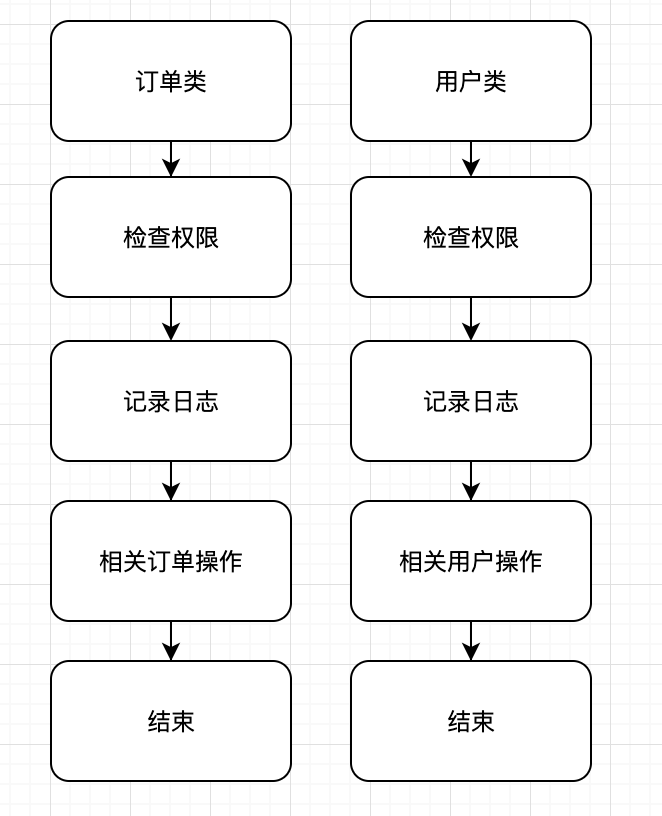
面向切面的程序设计（Aspect-oriented programming，AOP，又译作面向方面的程序设计、剖面導向程式設計）是计算机科学中的一种程序设计范型，旨在将横切关注点与业务主体进行进一步分离，以提高程序代码的模块化程度。 --《维基百科》

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。 --《百度百科》

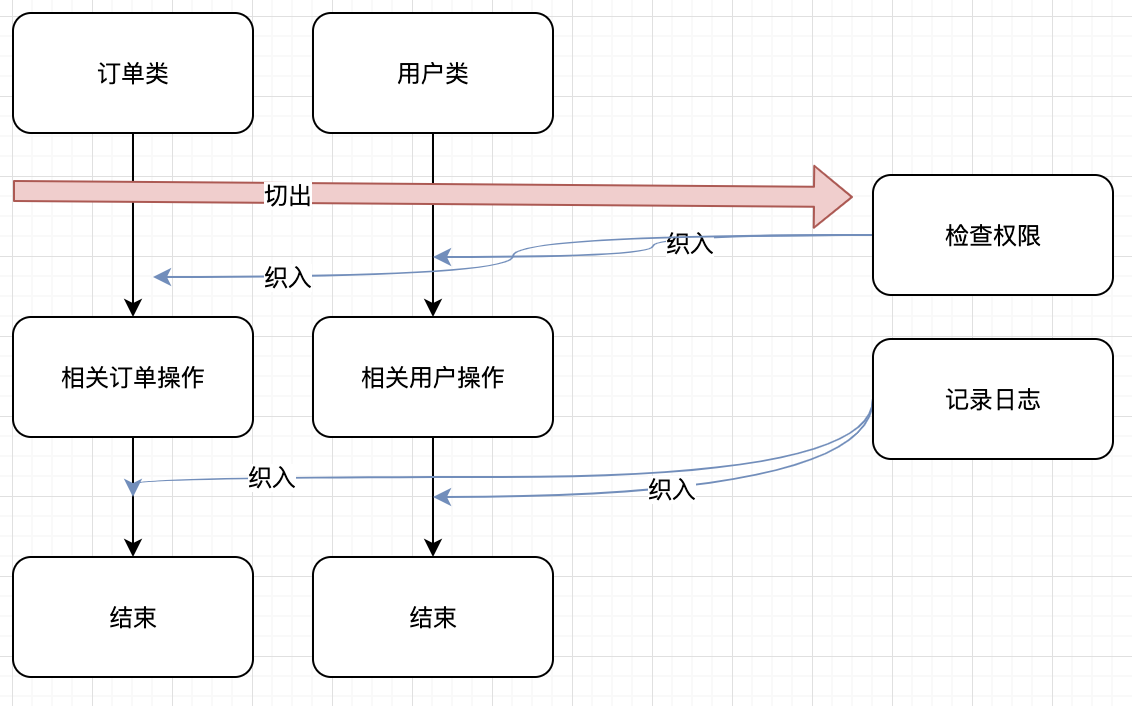
AOP 的主要作用就是在不侵入原有功能代码的情况下，给原有的功能添加新的功能。

我们都知道 OOP 实际上就是对我们的功能属性，方法做一个抽象封装，能够比较清晰的划分逻辑单元，但是 OOP 只能够进行纵向的抽象封装，无法很好的解决横向的重复问题，而 AOP 则很好的解决了这一问题，下面请看以下图示。

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/aop/OOP.png )



如上图我们可以看到我们有两个类 订单类 , 用户类 我们都相关功能做了封装，但是遇到类似 权限检查，日志记录这种重复性的操作却没有太好的办法去解决它，而 AOP 的思想就是将这些功能横向切出去，再在适当的时候将这些功能织入进来，如图( https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/aop/AOP.png )。



如图所示这就是 AOP 所做的功能。AOP 和 OOP 一样是一种思想，它同 OOP 一样也有一些相关的专业术语，下面我会向大家介绍这些术语。

//

///2/ 相关术语解释

首先我们先将相关术语进行一个大致的解释。

//a/ 通知（Advice）

就是你想要的功能，用方法实现的功能，也就是上面说的安全、事务、日志等。你先把功能用方法给先定义好，然后在想用的地方用一下。

//

//b/ 连接点（JoinPoint）

就是允许你把通知（Advice）放在的地方，基本每个方法的前、后（两者都有也行），或抛出异常时都可以是连接点，Swoft 只支持方法级的连接点。只要记住，和方法有关的前前后后都是连接点。

//

//c/ 切入点（Pointcut）

这个就是定义要切入的具体的方法，为什么呢，假设一下，一个类中有10个方法，每个方法都有好多个连接点(JoinPoint),但是你并不想在所有方法都使用通知（织入），你只想织入其中的某几个方法，那么就是用这个切入点来定义具体的方法。

//

//d/ 切面（Aspect）

是通知（Advice）和切入点 （Pointcut）的结合。通知说明了干什么，切入点说明了在哪里干，一般情况下连接点是在指定切入点的时候就指定好了的，上面单独把连接点写出来只是为了让其更好理解。

//

//c/ 引入（introduction）

就是将我们新的类和方法，用到目标类中。

//

//d/ 目标（target）

引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑，他可以在毫不知情的情况下，被织入切面。

//

//e/ 代理（proxy）

整套AOP机制的，都是通过代理，Swoft 使用了 PHP-Parse 类库来更方便的实现AOP。

//

//f/ 织入（weaving）

把切面应用到目标对象来创建新的代理对象的过程。

//

///3/ 如何使用AOP

这里主要是介绍如何在SWOFT 中使用 AOP。

在SWOFT 中使用AOP 特别的方便，我们只需要通过 注解 的方式来声明一个 切面类 就可以了，这里这是简单的演示，关于 AOP 类的详细声明相关的说明，请看后续 声明( https://www.swoft.org/docs/2.x/zh-CN/aop/index.html ) 章节。

<?php declare(strict\_types=1);

namespace App\Aspect;

use Swoft\Aop\Annotation\Mapping\Aspect;

use Swoft\Aop\Annotation\Mapping\Before;

use Swoft\Aop\Annotation\Mapping\PointBean;

/\*\*

 \* AOP切面类

 \*

 \* @since 2.0

 \*

 \* 声明切面类

 \* @Aspect(order=1)

 \*

 \* 使用注解定义切入点

 \* @PointBean(include={"App\Http\Controller\TestExecTimeController"})

 \*/

class CalcExecTimeAspect

{

    /\*\*

     \* 定义通知链接点

     \* @Before()

     \*/

    public function before()

    {

        //TODO::这里做具体的通知（Advice）

    }

}

?>

//

(2)、使用 Swoft 的 AOP 完成上方案例

至此我们重新回顾下本章开篇提到的场景案例，这里我们使用 Swoft 框架的AOP 来快速的实现案例中的需求。

<?php declare(strict\_types=1);

namespace App\Aspect;

use Swoft\Aop\Annotation\Mapping\Aspect;

use Swoft\Aop\Annotation\Mapping\Before;

use Swoft\Aop\Annotation\Mapping\After;

use Swoft\Aop\Annotation\Mapping\PointBean;

/\*\*

 \* AOP切面类

 \*

 \* @since 2.0

 \*

 \* 声明切面类

 \* @Aspect(order=1)

 \*

 \* 使用注解定义切入点

 \* @PointBean(include={"OrderServer::class"})

 \*/

class CalcExecTimeAspect

{

    /\*\*

     \* 定义通知链接点

     \* @Before()

     \*/

    public function before()

    {

        //TODO::在这里我们记录接口调用时传过来的参数

    }

     /\*\*

      \* 定义通知链接点

      \* @AfterReturning()

      \*/

     public function afterReturning()

     {

         //TODO::这里可以处理方法返回的结果

     }

     /\*\*

       \* 定义通知链接点

       \* @AfterThrowing()

       \*/

      public function afterThrowing()

      {

          //TODO::如果有出错异常，在这里我们处理出错的日志记录等操作

      }

}

?>

如上，利用AOP 我们可以很快速的完成此需求。更完整的使用案例请查看 使用( https://www.swoft.org/docs/2.x/zh-CN/aop/usage.html ) 章节。

--

1、声明

( https://www.swoft.org/docs/2.x/zh-CN/aop/statement.html )

//

(1)、声明切面

Aspect 类与任何其他正常的 bean 类似，并且可能像任何其他类一样拥有方法和字段，但它们将使用 @Aspect 注释，如下所示。

use Swoft\Aop\Annotation\Mapping\Aspect;

/\*\*

 \* Class DemoAspect

 \*

 \* @since 2.0

 \*

 \* @Aspect(order=1)

 \*/

class DemoAspect

{

    // ...

}

//

@Aspect

定义一个类为切面类，定义一个类为切面类。

//

(2)、声明切入点

一个切入点有助于确定不同的意见执行感兴趣的连接点（即方法）。

/\*\*

 \* Class DemoAspect

 \*

 \* @since 2.0

 \*

 \* @Aspect(order=1)

 \*

 \* @PointBean(

 \*     include={"testOrderAop"}

 \* )

 \*/

class DemoAspect

{

    // ...

}

//

@PointBean

定义bean切入点, 这个bean类里的所有public方法执行都会经过此切面类的代理：include 定义需要切入的实体名称集合；exclude 定义需要排除的实体名称集合。注意 实体名称(类名)必须指定 namespace 完整路径，例如 'App\Controller\HomeController' 或者先用 use 将目标类 use 进来，示例：@PointBean(include={App\Controller\HomeController::class})。

//

@PointAnnotation

定义注解类切入点, 所有包含使用了对应注解的方法都会经过此切面类的代理，include 定义需要切入的注解类名集合，exclude 定义需要排除的注解类名集合。注意 实体名称(类名)必须指定 namespace 完整路径 例如 'App\Controller\HomeController' 或者先用 use 将目标类 use 进来，示例：@PointAnnotation(include={Swoft\Http\Server\Annotation\Mapping\RequestMapping::class})。

//

@PointExecution

定义匹配切入点, 指明要代理目标类的哪些方法：include 定义需要切入的匹配集合，匹配的类方法，支持正则表达式；exclude 定义需要排序的匹配集合，匹配的类方法，支持正则表达式。注意 实体名称(类名)必须指定 namespace 完整路径 例如 'App\Controller\HomeController' 或者先用 use 将目标类 use 进来。示例：@PointExecution(include={"App\Http\Controller\AspectTestController::a.\*"})。这里需要注意下，如果需要使用正则，则传入的必须使用双引号 " " 引起来，命名空间分隔符必须使用 \ 转义，同时双引号内必须是类的完整路径。

@PointBean、@PointAnnotation、@PointExecution 三种定义的关系是并集，三种里面定义的排除也是并集后再排除。建议为了便于理解和使用，一个切面类尽量只使用上面三个中的一个。

//

(3)、声明通知

use Swoft\Aop\Annotation\Mapping\After;

use Swoft\Aop\Annotation\Mapping\AfterReturning;

use Swoft\Aop\Annotation\Mapping\AfterThrowing;

use Swoft\Aop\Annotation\Mapping\Around;

use Swoft\Aop\Annotation\Mapping\Aspect;

use Swoft\Aop\Annotation\Mapping\Before;

use Swoft\Aop\Annotation\Mapping\PointBean;

use Swoft\Aop\Point\JoinPoint;

use Swoft\Aop\Point\ProceedingJoinPoint;

/\*\*

 \* Class DemoAspect

 \*

 \* @since 2.0

 \*

 \* @Aspect(order=1)

 \*

 \* @PointBean(

 \*     include={"testOrderAop"}

 \* )

 \*/

class DemoAspect

{

    /\*\*

     \* @Before()

     \*/

    public function before()

    {

        // before

    }

    /\*\*

     \* @After()

     \*/

    public function after()

    {

        // After

    }

    /\*\*

     \* @AfterReturning()

     \*

     \* @param JoinPoint $joinPoint

     \*

     \* @return mixed

     \*/

    public function afterReturn(JoinPoint $joinPoint)

    {

        $ret = $joinPoint->getReturn();

        // After return

        return $ret;

    }

    /\*\*

     \* @Around()

     \*

     \* @param ProceedingJoinPoint $proceedingJoinPoint

     \*

     \* @return mixed

     \*/

    public function around(ProceedingJoinPoint $proceedingJoinPoint)

    {

        // Before around

        $result = $proceedingJoinPoint->proceed();

        // After around

        return $result;

    }

    /\*\*

     \* @param \Throwable $throwable

     \*

     \* @AfterThrowing()

     \*/

    public function afterThrowing(\Throwable $throwable)

    {

        // afterThrowing

    }

}

@Before 前置通知，在目标方法执行前先执行此方法。

@After 后置通知，在目标方法执行后执行此方法。

@AfterReturning 最终返回通知。

@AfterThrowing 异常通知，在目标方法执行抛出异常时执行此方法。

@Around 环绕通知，在目标方法执行前、后都执行此方法。

//

(4)、AOP 注意事项

AOP只拦截 public 和 protected 方法，不拦截private方法。

另外在 Swoft AOP 中 如果切入了多个方法，此时在某一个方法内调用了另一个被切入的方法，此时AOP 也会织入通知。 例如 我们定义了一个类 A 它有两个 public 方法 fun1(),fun2(),然后我们定义一个切面，使用 @PointBean(include={A::class}) 注解将 A 类中的两个方法都进行切入，这是我们看下这两个方法的定义。

<?php

class A

{

    function fun1()

    {

        echo 'fun1'."\n";

    }

    function fun2()

    {

        $this->fun1();

        echo 'fun2'."\n";

    }

}

?>

此时如果我们访问 fun2() 这个方法，那么我们的切面就会执行两次,切面的执行顺序和方法的执行顺序相同。先执行 fun2() 方法的织入通知，再执行 fun1() 方法的织入通知。

--

2、使用

( https://www.swoft.org/docs/2.x/zh-CN/aop/usage.html )

本章以计算类里面每个方法执行时间为例，简单介绍使用切面编程。

//

(1)、编写测试类

//// 首先先编写一个Http Controller 类

// App\Http\Controller\TestExecTimeController

<?php

namespace App\Http\Controller;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

/\*\*

 \* 运算时间测试类

 \*

 \* @Controller()

 \*/

class TestExecTimeController

{

    /\*\*

     \* 闭包递归 计算阶乘

     \*

     \* @RequestMapping(route="test/{number}")

     \*

     \* @param int $number

     \*

     \* @return array

     \*/

    public function factorial(int $number): array

    {

        $factorial = function ($arg) use (&$factorial) {

            if ($arg == 1) {

                return $arg;

            }

            return $arg \* $factorial($arg - 1);

        };

        return [$factorial($number)];

    }

    /\*\*

     \* 计算1～1000的和，最后休眠1s

     \*

     \* @RequestMapping()

     \*/

    public function sumAndSleep(): array

    {

        $sum = 0;

        for ($i = 1; $i <= 1000; $i++) {

            $sum = $sum + $i;

        }

        usleep(1000);

        return [$sum];

    }

}

?>

//

(2)、编写切面类

创建一个切面类，对我们的测试类进行切入。

//

//// App\Aspect\CalcExecTimeAspect

<?php declare(strict\_types=1);

namespace App\Aspect;

use Swoft\Aop\Annotation\Mapping\After;

use Swoft\Aop\Annotation\Mapping\Aspect;

use Swoft\Aop\Annotation\Mapping\Before;

use Swoft\Aop\Annotation\Mapping\PointBean;

use Swoft\Aop\Point\JoinPoint;

/\*\*

 \* AOP切面类

 \*

 \* @since 2.0

 \*

 \* 声明切面类

 \* @Aspect(order=1)

 \*

 \* 声明为 PointBean 类型的切面

 \* @PointBean(include={"App\Http\Controller\TestExecTimeController"})

 \*/

class CalcExecTimeAspect

{

    protected $start;

    /\*\*

     \* 定义通知点

     \* @Before()

     \*/

    public function before()

    {

        $this->start = microtime(true);

    }

    /\*\*

     \* 定义通知点

     \* @After()

     \*/

    public function after(JoinPoint $joinPoint)

    {

        $method = $joinPoint->getMethod();

        $after = microtime(true);

        $runtime = ($after - $this->start) \* 1000;

        echo "{$method} 方法，本次执行时间为: {$runtime}ms\n";

    }

}

?>

//

//// 启动 swoft 之后正常的访问我们的测试控制器，将会在控制器看到输出。

//1/ 访问控制器

$ curl http://localhost:18306/testExecTime/test/10

[3628800]%

$ curl http://localhost:18306/testExecTime/sumAndSleep

[500500]%

//

//2/ 控制台输出

factorial 方法，本次执行时间为: 0.10013580322266ms

--

3、顺序

( https://www.swoft.org/docs/2.x/zh-CN/aop/order.html )

多个切面执行是按照 order 值越小越先执行，一个切面多个通知点，也是按照一定的顺序执行的。

//

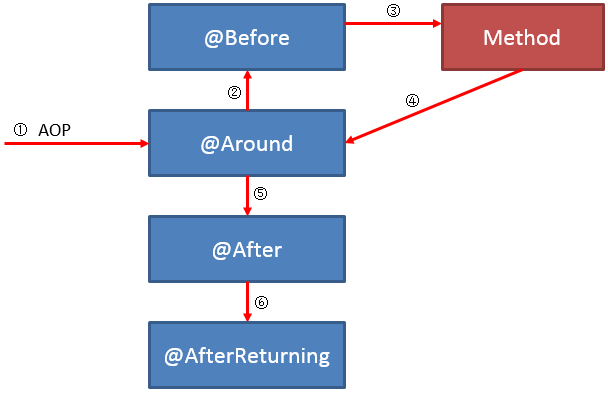
(1)、单切面

一个切面顺序相对来说很简单，但是分异常和正常两种情况。

//

///1/ 正常顺序

( https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/aop/1.jpg )



|1>@Around 通知的 before around 业务。

|2>@Before 通知。

|3>method 方法。

|4>@Around 通知的 after around 业务。

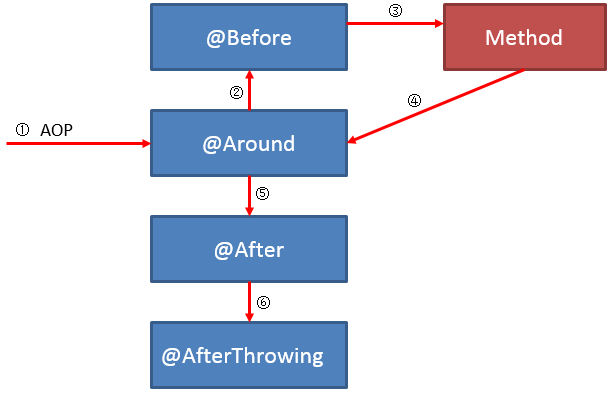
|5>@After 通知。

|6>执行 @AfterReturn 通知。

//

///2/ 异常顺序

( https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/aop/2.jpg )



|1>@Around 通知的 before around 业务。

|2>@Before 通知。

|3>method 方法。

|4>@Around 通知的 after around 业务。

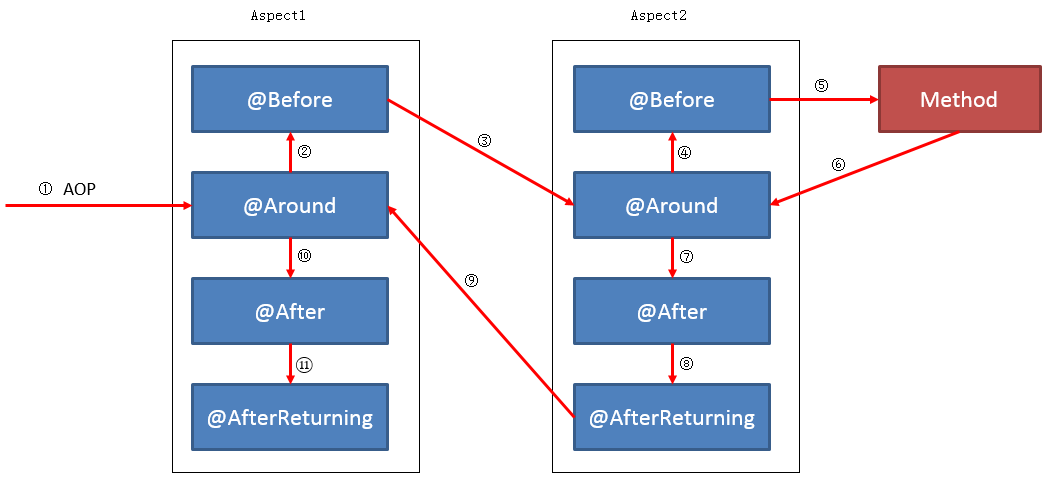
|5>@After 通知。

|6>执行 @AfterThrowning 通知。

//

(2)、多切面

( https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/aop/3.jpg )



//// 多切面以正常情况为例

|1>Aspect1 @Around 通知的 before around 业务。

|2>Aspect1 @Before 通知。

|3>Aspect2 @Around 通知的 before around 业务。

|4>Aspect2 @Before 通知。

|5>method 方法。

|6>Aspect2 @Around 通知的 after around 业务。

|7>Aspect2 @After 通知。

|8>Aspect2 执行 @AfterReturn 通知。

|9>Aspect1 @Around 通知的 after around 业务。

|10>Aspect1 @After 通知。

|11>Aspect1 执行 @AfterReturn 通知。

--

六、错误处理

( https://www.swoft.org/docs/2.x/zh-CN/error/index.html )

swoft 提供了完善的异常与错误处理机制，与通常fpm下情况不同，swoft里将错误分为了不同的场景类型。

//

0、错误处理

//

(1)、安装

composer require swoft/error

//

(2)、错误场景

场景类型的划分：主要是根据swoole的回调事件范围来划分。在不同的场景里，即使是同一个地方抛出的异常或错误，处理方式可能也是不同的。

比如：db里抛出了一个 DbException，在 http 运行场景里你需要处理后返回一个 Reponse 对象。而在cli等环境里，你可以只处理无需返回任何结果。

//

(3)、错误场景类型

在 Swoft\Error\ErrorType 定义了swoft划分的场景类型。

//

<?php declare(strict\_types=1);

namespace Swoft\Error;

/\*\*

 \* Class ErrorType

 \*

 \* @since 2.0

 \*/

final class ErrorType

{

    public const CLI  = 2;

    public const RPC  = 3;

    public const UDP  = 4;

    public const TCP  = 5;

    public const SOCK = 7;

    public const TASK = 8;

    // HTTP server

    public const HTTP = 16;

    // WebSocket server

    public const WS\_HS  = 21;

    public const WS\_OPN = 22;

    public const WS\_MSG = 23;

    public const WS\_CLS = 24;

    public const SYS = 85;

    // default

    public const DEF     = 90;

    public const DEFAULT = 90;

}

?>

//

(4)、Git仓库

Github https://github.com/swoft-cloud/swoft-error

//

(5)、参与贡献

欢迎参与贡献，您可以：

|1>fork 我们的开发仓库 swoft/component ( https://github.com/swoft-cloud/swoft-component )。

|2>修改代码然后发起 PR。

|3>关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、添加处理器 ( 异常处理使用 )

由前面的介绍我们知道，swoft里的异常处理是划分场景的。在几个通用的server里，我们已经内置了对应场景的基础异常处理类。

(1)、注解

由 swoft-error 组件提供的注解标签 Swoft\Error\Annotation\Mapping\ExceptionHandler 提供异常处理器类的标记。

//// ExceptionHandler 注解

@ExceptionHandler 标记一个类为异常处理器。拥有属性：exceptions 定义需要处理的异常类，需完整类名。

//

(2)、匹配逻辑

在同一个场景里，可以定义多个处理器来处理不同的异常：在发生异常时，首先会使用异常类完整匹配来查找处理器，匹配成功就由它处理；完整匹配失败，则会检查异常类是否是已注册的异常类的子类，是就选择第一个匹配的处理器处理；仍然匹配失败，则会由系统默认处理。

//

(3)、Http Request 异常处理

你的异常类需要继承 Swoft\Http\Server\Exception\Handler\AbstractHttpErrorHandler。在处理异常后你必须返回一个 Swoft\Http\Message\Response 对象作为对http客户端的响应。

//// 使用示例

// 这个示例代码来自 swoft/swoft 的 app/Exception/Handler/HttpExceptionHandler。

<?php declare(strict\_types=1);

namespace App\Exception\Handler;

use const APP\_DEBUG;

use function get\_class;

use ReflectionException;

use function sprintf;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Error\Annotation\Mapping\ExceptionHandler;

use Swoft\Http\Message\Response;

use Swoft\Http\Server\Exception\Handler\AbstractHttpErrorHandler;

use Throwable;

/\*\*

 \* Class HttpExceptionHandler

 \*

 \* @ExceptionHandler(\Throwable::class)

 \*/

class HttpExceptionHandler extends AbstractHttpErrorHandler

{

    /\*\*

     \* @param Throwable $e

     \* @param Response   $response

     \*

     \* @return Response

     \* @throws ReflectionException

     \* @throws ContainerException

     \*/

    public function handle(Throwable $e, Response $response): Response

    {

        // Debug is false

        if (!APP\_DEBUG) {

            return $response->withStatus(500)->withContent(

                sprintf(' %s At %s line %d', $e->getMessage(), $e->getFile(), $e->getLine())

            );

        }

        $data = [

            'code'  => $e->getCode(),

            'error' => sprintf('(%s) %s', get\_class($e), $e->getMessage()),

            'file'  => sprintf('At %s line %d', $e->getFile(), $e->getLine()),

            'trace' => $e->getTraceAsString(),

        ];

        // Debug is true

        return $response->withData($data);

    }

}

?>

//

(4)、RPC 异常处理

你的异常类需要继承 Swoft\Rpc\Server\Exception\Handler\AbstractRpcServerErrorHandler。在处理异常后你必须返回一个 Swoft\Rpc\Server\Response 对象作为对rpc客户端的响应。

//

//// 使用示例

// 这个示例代码来自 swoft/swoft 的 app/Exception/Handler/RpcExceptionHandler。

<?php declare(strict\_types=1);

namespace App\Exception\Handler;

use ReflectionException;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Error\Annotation\Mapping\ExceptionHandler;

use Swoft\Log\Debug;

use Swoft\Rpc\Error;

use Swoft\Rpc\Server\Exception\Handler\RpcErrorHandler;

use Swoft\Rpc\Server\Response;

use Throwable;

/\*\*

 \* Class RpcExceptionHandler

 \*

 \* @since 2.0

 \*

 \* @ExceptionHandler(\Throwable::class)

 \*/

class RpcExceptionHandler extends RpcErrorHandler

{

    /\*\*

     \* @param Throwable $e

     \* @param Response  $response

     \*

     \* @return Response

     \* @throws ReflectionException

     \* @throws ContainerException

     \*/

    public function handle(Throwable $e, Response $response): Response

    {

        // Debug is false

        if (!APP\_DEBUG) {

            $message = sprintf(' %s At %s line %d', $e->getMessage(), $e->getFile(), $e->getLine());

            $error   = Error::new($e->getCode(), $message, null);

        } else {

            $error = Error::new($e->getCode(), $e->getMessage(), null);

        }

        Debug::log('Rpc server error(%s)', $e->getMessage());

        $response->setError($error);

        // Debug is true

        return $response;

    }

}

?>

//

//// 使用说明

class BusinessLogic

{

    public function doSomething()

    {

        throw new BusinessException("Error Processing Request", 500);

    }

}

//

(5)、结语

当这里抛出异常：在http request 场景范围内，它会被 HttpExceptionHandler 处理；若是在RPC server 请求场景里，则会由 RpcExceptionHandler 处理。

上面的处理器是通过查找异常父类找到的，当然你也可以定义一个针对 BusinessException 的异常处理器 @ExceptionHandler(BusinessException::class)。

通过上面的示例我们可以看到，即使你是同一个地方抛出的异常，只要你定义了不同场景的异常处理器。 就可以分别针对不同场景的请求(如上面的 http 和 rpc)，在里面返回不同的响应，而无需做额外的检查与判断。更多场景的异常处理也可以参考上面的示例来编写。

--

--

七、公共方法 ( 公共函数 )

( https://www.swoft.org/docs/2.x/zh-CN/common/index.html )

本章会详细介绍框架内置封装的一些函数，开发者在实际业务中可以直接调用，不用再去封装。协程函数、字符串函数、数组函数、验证函数。

//

1、协程方法 ( 协程函数 )

( https://www.swoft.org/docs/2.x/zh-CN/common/co.html )

框架中封装了一些协程操作的函数，方便开发者调用。

//

(1)、创建协程

特别说明：Swoft 框架中一定不要直接使用 Swoole 提供的 `go` 函数创建协程，否则会导致请求和上下文丢失导致一些奇葩问题。

use Swoft\Co;

Co::create(function(){

    // to do

});

如上是框架中创建协程的原型，框架也提供了更为简单的 sgo 函数来创建协程，替换 Swoole 的 go 函数使用。

sgo(function(){

    // todo

});

`sgo` 函数使用和 Swoole `go` 函数完全一样，切记框架中只能使用 `sgo` 函数，不能直接使用 `go` 函数。

//

(2)、协程ID

获取当前协程ID, -1 非协程环境。

use Swoft\Co;

$id = Co::id();

//

(3)、顶级协程ID

获取顶级(最外层协程ID)。

use Swoft\Co;

$id = Co::tid();

//

(4)、读取文件

全量读取文件。

public static function readFile(string $filename): string

//// 读取的文件路径

读取成功返回字符串内容，读取失败返回 false，可使用swoole\_last\_error获取错误信息。readFile方法没有尺寸限制，读取的内容会存放在内存中，因此读取超大文件时可能会占用过多内存。

//

//// 使用实例

use Swoft\Co;

$fileName = 'test.file';

$data = Co::readFile($fileName);

//

(5)、写文件

public static function writeFile(string $filename, string $data, int $flags = null): int

filename为文件的名称，必须有可写权限，文件不存在会自动创建。打开文件失败会立即返回false。

$fileContent为要写入到文件的内容，最大可写入4M。

flags为写入的选项，默认会清空当前文件内容，可以使用FILE\_APPEND表示追加到文件末尾。

//

//// 写入成功返回true，写入失败返回false

use Swoft\Co;

$fileName = 'test.file';

$data = Co::writeFile($fileName, 'data');

//

(6)、并发

框架底层通过协程通道，封装了一套多混合IO并发操作的方法，一般用于多个流程没有依赖，可以并发执行，提高执行效率。

public static function multi(array $requests, float $timeout = 0): array

参数requests 多个操作集合，KV 数组格式。

参数timeout 超时时间，默认永久超时。

//

并发执行结果，按照 requests 集合数组 key 对应关系返回，如果一个 key 对应的值返回 false, 意味着该操作执行失败。requests 里面的每个操作可以执行的业务无上限，根据自己的业务而定。

//

//// requests 格式支持多种方式

对象的某个方法；

对象的静态方法；

闭包匿名函数；

//

//// 使用实例

use Swoft\Co;

/\*\*

 \* Class CoTest

 \*

 \* @since 2.0

 \*/

class CoTest

{

    /\*\*

     \* @throws \ReflectionException

     \* @throws \Swoft\Bean\Exception\ContainerException

     \*/

    public function testMulti()

    {

        $requests = [

            'method'       => [$this, 'requestMethod'],

            'staticMethod' => "SwoftTest\Unit\CoTest::requestMehtodByStatic",

            'closure'      => function () {

                $cli = new Client('www.baidu.com', 80);

                $cli->get('/');

                $result = $cli->body;

                $cli->close();

                return $result;

            }

        ];

        $response = Co::multi($requests);

    }

    /\*\*

     \* @return mixed

     \*/

    public function requestMethod()

    {

        $cli = new Client('www.baidu.com', 80);

        $cli->get('/');

        $result = $cli->body;

        $cli->close();

        return $result;

    }

    /\*\*

     \* @return mixed

     \*/

    public static function requestMehtodByStatic()

    {

        $cli = new Client('www.baidu.com', 80);

        $cli->get('/');

        $result = $cli->body;

        $cli->close();

        return $result;

    }

}

?>

--

2、定时器

( https://www.swoft.org/docs/2.x/zh-CN/common/timer.html )

//

(1)、Tick (定时器)

对应一些延迟执行或者周期性执行的任务，此时需要使用定时器实现。Swoft 框架中封装了一套定时器操作方式，开发者可以直接使用，切记一定要使用 Swoft 框架封装的，如果直接使用 Swoole 提供的会因为上下文，带来一系列问题。

特别说明：2.0.5+ 支持，定时器一定要使用 Swoft 框架封装的，不要直接使用 Swoole 提供的。

//

//// 周期性的执行一项任务，使用例子如下方式；

$paramOne = 1;

$paramTwo = 1;

\Swoft\Timer::tick(1, function ($paramOne, $paramTwo) {

    // To to something

}, $paramOne, $paramTwo);

//

//// \Swoft\Timer::tick 详细参数

$msec 周期执行的时间，单位是毫秒。

$callback 回调函数。

$params 传递的参数，会原封不动的传递给回调函数。

//

(2)、After

//// 延迟一段时间执行任务，使用例子如下方式；

$paramOne = 1;

$paramTwo = 1;

\Swoft\Timer::after(1, function ($paramOne, $paramTwo) {

    // To to something

}, $paramOne, $paramTwo);

/// 参数

$msec 延迟时间，单位是毫秒。

$callback 回调函数。

$params 传递的参数，会原封不动的传递给回调函数。

--

3、通用函数

( http://swoft.org/docs/2.x/zh-CN/common/generic.html )

(1)、env

//// 获取环境变量的值

function env(string $key = null, $default = null): mixed

//

(2)、alias

//// 获取路径别名的值

function alias(string $key): string

//

(3)、bean

//// 从容器中获取bean对象，等同于 \Swoft::getBean()

function bean(string $key): object

//

(4)、config

//// 获取应用配置的值

function config(string $key, $default = null): mixed

//

(5)、sgo

//// 开启一个新的协程

function sgo(callable $callable, bool $wait = true): void

//

(6)、server

function server(): \Swoft\Server\Server

//

(7)、context

//// 获取上下文对象

function context(): ContextInterface

//

(8)、container

//// 获取容器对象

function container(): Container

--

4、基础帮助库

(1)、String <无>

(2)、Array <无>

--

5、数组

为了方便开发swoft官方提供了数组的基础帮助库，大大提升开发效率。以下示例代码可以直接运行。

//

(1)、toArray

//// 将对象或对象数组转换为数组

public static function toArray($object, $properties = [], $recursive = true): array

$object:object|array|string；要转换为数组的对象。

$properties:array；从对象类名称到需要放入结果数组的属性的映射。

$recursive:boolean；是否以递归方式将对象属性转换为数组（默认是）。

返回值：array（对象的数组表示）。

//

//// 示例

<?php

$objSub = new \stdClass();

$objSub->version = '2.0.6';

$objSub->url = 'https://www.swoft.org';

$obj = new \stdClass();

$obj->name = 'swoft framework 2.x';

$obj->desc = $objSub;

print\_r(ArrayHelper::toArray($obj));//['name'=>'swoft framework 2.x','desc'=>['version'=>'2.0.6','url'=>'https://www.swoft.org']]

print\_r(ArrayHelper::toArray($obj, [get\_class($obj) => ['name']]));//['name'=>'swoft framework 2.x']

print\_r(ArrayHelper::toArray($obj,[],false));

/\*

Array

(

    [name] => swoft framework 2.x

    [desc] => stdClass Object

        (

            [version] => 2.0.6

            [url] => https://www.swoft.org

        )

)

\*/

?>

//

(2)、merge

///1/ 功能

都有一个具有相同字符串键值的元素，后者将覆盖前者（与array\_merge\_recursive不同）。如果两个数组都具有数组类型的元素且具有相同的键，则将进行递归合并。对于整数键控元素，后一个数组中的元素将附加到前一个数组。

//

///2/ 概要

public static function merge($a, $b): array

//

///3/ 参数说明

$a:array；要合并的数组。

$b:array；要合并的数组。

//

///4/ 返回值

array（数组合并的数组（原始数组不会更改））

//

///5/ 示例

<?php

$a = ['s'=>1,'w'=>'o','f'=>[9501,9502],'t'=>'swoft',666];

$b = ['t'=>'swoft 2.x',2,3];

print\_r(ArrayHelper::merge($a,$b));//['s'=>1,'w'=>'o','f'=>[0=>9501,1=>9502],'t'=>'swoft 2.x',0=>666,1=>2,2=>3]

?>

//

(3)、getValue

///1/ 功能

使用给定的键或属性名称检索数组元素或对象属性的值。如果数组或对象中不存在该键，则将返回默认值。

//

///2/ 概要

public static function getValue($array, $key, $default = null)

//

///3/ 参数

$array:array|object；要从中提取之的数组或对象。

$key:string|Closure|array；数组的键名，一个数组的键或者对象的属性名，或一个返回值的匿名函数。这个匿名函数的签名应该是这样的function($array,$defaultValue)。

$default:mixed；如果指定的数组键不存在，则返回默认值。从对象获取值时不使用。

//

///4/ 返回值

mixed（找到的元素的值，否则为默认值）。

//

///5/ 示例

<?php

$a = ['s'=>123,'w'=>456];

print\_r(ArrayHelper::getValue($a,'s'));//123

print\_r(ArrayHelper::getValue($a,'o'));//null

print\_r(ArrayHelper::getValue($a,'o','789'));//789

$obj = new \stdClass();

$obj->name='swoft framework';

$obj->version= '2.x';

$obj->desc = ['type'=>'coroutine','fpm'=>'n'];

print\_r(ArrayHelper::getValue($obj,function ($obj,$defaultValue){

    return $obj->name.' '.$obj->version;

}));//swoft framework 2.x

print\_r(ArrayHelper::getValue($obj,'name'));//swoft framework

print\_r(ArrayHelper::getValue($obj,'desc.fpm'));//n

?>

//

(4)、remove

///1/ 功能

从数组中删除项并返回值。如果数组中不存在该键，则将返回默认值。

//

///2/ 概要

public static function remove(&$array, $key, $default = null)

//

///3/ 参数说明

$array:array；从中提取值的数组。

$key:string；数组元素的键名。

$default:mixed；如果指定的数组键不存在，这个默认值将被返回。

//

///4/ 返回值

mixed|null（找到的元素的值，否则为默认值）

//

///5/ 示例

<?php

$a = ['s'=>1,'w'=>2];

ArrayHelper::remove($a,'s');

print\_r($a);//['w'=>2]

$b = ['s'=>3,'w'=>4];

ArrayHelper::remove($b,'o');

print\_r($b);//['s'=>3,'w'=>4]

$c = ['s'=>5,'w'=>6];

$ret = ArrayHelper::remove($c,'o',[7,8]);

print\_r($ret);//[7,8]

?>

//

(5)、except

///1/ 功能

获取除指定数组键之外的所有给定数组。

//

///2/ 摘要

public static function except($array, $keys): array

//

///3/ 参数说明

$array:array；

$keys:array|string；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['a' => 1, 'b' => 2, 'c' => ['d' => 1, 'e' => 2]];

$ret = ArrayHelper::except($arr, ['a']);

print\_r($ret);//['b' => 2, 'c' => ['d' => 1, 'e' => 2]];

$ret2 = ArrayHelper::except($arr, ['a', 'b', 'c.d']);

print\_r($ret2);//['c' => ['e' => 2]]

?>

//

(6)、forget

//

///1/ 功能

使用“点”表示法从给定数组中删除一个或多个数组项。

//

///2/ 摘要

public static function forget(&$array, $keys): void

//

///3/ 参数说明

$array:array；

$keys:array|string；

//

///4/ 返回值

void

//

///5/ 示例

$arr = ['a' => 1, 'b' => 2, 'c' => ['d' => 1, 'e' => 2]];

ArrayHelper::forget($arr, ['a']);

print\_r($arr);//['b' => 2, 'c' => ['d' => 1, 'e' => 2]]

$arr2 = ['a' => 1, 'b' => 2, 'c' => ['d' => 1, 'e' => 2]];

ArrayHelper::forget($arr2, ['a', 'b', 'c.d']);

print\_r($arr2);//['c' => ['e' => 2]]

//

(7)、pull

///1/ 功能

从数组中获取值，然后将其删除。

//

///2/ 摘要

public static function pull(&$array, $key, $default = null)

//

///3/ 参数说明

$array:array；

$key:string；

$default:mixed；

//

///4/ 返回值

mixed

//

///5/ 示例

<?php

$arr = ['a' => 1, 'b' => 2, 'c' => ['d' => 1, 'e' => 2]];

$rs = ArrayHelper::pull($arr, 'b');

print\_r($rs);//2

print\_r($arr);//['a' => 1, 'c' => ['d' => 1, 'e' => 2]]

$rs2 = ArrayHelper::pull($arr, 'g', 99);

print\_r($rs2);//99

?>

//

(8)、index

///1/ 功能

根据指定的键索引 和/或 对数组进行分组。输入应该是多维数组或对象数组。

$key可以是子数组的键名，对象的属性名，也可以是必须返回将用作键的值的匿名函数。

$groups是一个键数组，用于根据指定的键将输入数组分组到一个或多个子数组中。

如果$key被指定为null，或者除了未指定$groups之外，对应于该键的元素的值为null，则该元素被丢弃。

//

///2/ 摘要

public static function index($array, $key, $groups = []): array

//

///3/ 参数说明

$array:array；需要索引或分组的数组。

$key:string|Closure|null；列名或匿名函数，其结果将用于索引数组。

$groups:string|string[]|Closure[]|null；键数组，用于通过一个或多个键对输入数组进行分组。如果$key属性或其特定元素的值为null并且未定义$groups，则将丢弃该数组元素。否则，如果指定了$groups，则数组元素将添加到结果数组中而不包含任何键。

//

///4/ 返回值

array（对索引 和/或 分组数组进行排列）

//

///5/ 示例

<?php

$array = [

            ['id' => '123', 'Data' => 'abc', 'device' => 'laptop'],

            ['id' => '345', 'Data' => 'def', 'device' => 'tablet'],

            ['id' => '345', 'Data' => 'hgi', 'device' => 'smartphone'],

        ];

$result = ArrayHelper::index($array, 'id');

print\_r($result);

//['123'=>['id'=>'123','Data'=>'abc','device'=>'laptop'],'345'=>['id'=>'345','Data'=>'hgi','device'=>'smartphone']]

$result = ArrayHelper::index($array, function ($element) {

    return $element['id'];

});

print\_r($result);

//['123'=>['id'=>'123','Data'=>'abc','device'=>'laptop'],'345'=>['id'=>'345','Data'=>'hgi','device'=>'smartphone']]

$result = ArrayHelper::index($array, null, 'id');

print\_r($result);//['123'=>[0=>['id'=>'123','Data'=>'abc','device'=>'laptop']],'345'=>[0=>['id'=>'345','Data'=>'def','device'=>'tablet'],1=>['id'=>'345','Data'=>'hgi','device'=>'smartphone']]]

$result = ArrayHelper::index($array, 'Data', [function ($element) {

    return $element['id'];

}, 'device']);

print\_r($result);

//['123'=>['laptop'=>['abc'=>['id'=>'123','Data'=>'abc','device'=>'laptop']]],'345'=>['tablet'=>['def'=>['id'=>'345','Data'=>'def','device'=>'tablet']],'smartphone'=>['hgi'=>['id'=>'345','Data'=>'hgi','device'=>'smartphone']]]]

?>

//

(9)、getColumn

///1/ 功能

返回数组中指定列的值。输入数组应为多维或对象数组。

//

///2/ 摘要

public static function getColumn($array, $name, $keepKeys = true): array

//

///3/ 参数说明

$array:array；

$name:string|Closure；

$keepKeys:boolean；是否维护数组键。如果为false，则生成的数组将使用整数重新编制索引。

//

///4/ 返回值

array（列值列表）

//

///5/ 示例

<?php

$array = [

  ['id' => '123', 'Data' => 'abc'],

  ['id' => '345', 'Data' => 'def'],

];

$result = ArrayHelper::getColumn($array, 'id');//['123', '345']

$result2 = ArrayHelper::getColumn($array, function ($element) {

      return $element['id'];

});//['123','345']

?>

//

(10)、map

///1/ 功能

从多维数组或对象数组构建映射（键值对）。

$from和$to参数指定用于设置map的键名或属性名。

可选地，可以根据分组字段$group进一步对map进行分组。

//

///2/ 概要

public static function map($array, $from, $to, $group = null): array

//

///3/ 参数说明

$array:array

$from:string|Closure

$to:string|Closure

$group:string|Closure

//

///4/ 返回值

array

//

///5/ 示例

<?php

$array = [

  ['id' => '123', 'name' => 'aaa', 'class' => 'x'],

  ['id' => '124', 'name' => 'bbb', 'class' => 'x'],

  ['id' => '345', 'name' => 'ccc', 'class' => 'y'],

];

$result = ArrayHelper::map($array,'id','name');//['123'=>'aaa','124'=>'bbb','345'=>'ccc']

$result2 = ArrayHelper::map($array,'id','name','class');//['x'=>['123'=>'aaa','124'=>'bbb'],'y'=>['345'=>'ccc']]

?>

//

(11)、keyExists

///1/ 功能

检查给定数组是否包含指定的键。此方法通过支持不区分大小写的键比较来增强array\_key\_exists()函数。

//

///2/ 摘要

public static function keyExists($key, $array, $caseSensitive = true): ?bool

//

///3/ 参数说明

$key:string；检查的键。

$array:array；带有要检查的键的数组。

$caseSensitive:boolean；键比较是否应区分大小写。

//

///4/ 返回值

boolean（数组是否包含指定的键）

//

///5/ 示例

<?php

$arr = ['id' => 1, 'data' => 'a'];

$rs = ArrayHelper::keyExists('id', $arr);//true

$rs2 = ArrayHelper::keyExists('name', $arr);//false

?>

//

(12)、multisort

///1/ 功能

通过一个或多个键对一组对象或数组进行排序（具有相同的结构）。

//

///2/ 摘要

public static function multisort(&$array, $key, $direction = SORT\_ASC, $sortFlag = SORT\_REGULAR): void

//

///3/ 参数说明

$array:array；要排序的数组。调用此方法后，将修改该数组。

$key:string|Closure|array；要排序的key(s)。这指的是子数组元素的键名，对象的属性名，或返回值以进行比较的匿名函数。匿名函数签名应该是：function（$item）。要按多个键排序，在这里提供一组键。

$direction:integer|array；排序方向。它可以是SORT\_ASC或SORT\_DESC。使用具有不同排序方向的多个键进行排序时，请使用排序方向数组。

$sortFlag:integer|array；PHP排序标志。有效值包括SORT\_REGULAR，SORT\_NUMERIC，SORT\_STRING，SORT\_LOCALE\_STRING，SORT\_NATURAL和SORT\_FLAG\_CASE。有关详细信息，请参阅[PHP手册]（http://php.net/manual/en/function.sort.php）。使用具有不同排序标志的多个键进行排序时，请使用排序标志数组。

//

///4/ 返回值

void

//

///5/ 异常

如果$direction或$sortFlag参数的元素数与$key的元素数不正确，则返回InvalidArgumentException。

//

///6/ 示例

<?php

$arr = [

    ['id' => 1,'age'=>33],

    ['id' => 5,'age'=>45],

    ['id' => 1,'age'=>41],

];

ArrayHelper::multisort($arr,function ($item){

    return $item['age'];

},SORT\_DESC);

print\_r($arr);//[['id'=>5,'age'=>45],['id'=>1,'age'=>41],['id'=>1,'age'=>33]]

$arr2 = [

    ['id' => 1,'age'=>33],

    ['id' => 5,'age'=>45],

    ['id' => 1,'age'=>41],

];

ArrayHelper::multisort($arr2, ['id','age'],SORT\_DESC);

print\_r($arr2);//[['id'=>5,'age'=>45],['id'=>1,'age'=>41],['id'=>1,'age'=>33]]

?>

//

(13)、isAssociative

///1/ 功能

返回一个值，指示给定数组是否为关联数组。如果数组的所有键都是字符串，则该数组是关联的。如果$allStrings为false，那么如果一个数组中至少有一个是一个字符串，则该数组将被视为关联数组。请注意，空数组不会被视为关联数组。

//

///2/ 概要

public static function isAssociative($array, $allStrings = true): ?bool

//

///3/ 参数说明

$array:array；正在检查的数组。

$allStrings:boolean；

//

///4/ 返回值

boolean（数组是否是关联的）

//

///5/ 示例

<?php

$arr = ['id' => 1, 'data' => 'a'];

$rs = ArrayHelper::isAssociative($arr);//true

$arr2 = [1 => 1, 'data' => 'a'];

$rs2 = ArrayHelper::isAssociative($arr2);//false

$rs3 = ArrayHelper::isAssociative($arr2, false);//true

?>

//

(14)、isIndexed

///1/ 功能

返回一个值，指示给定数组是否为索引数组。

如果数组的所有键都是整数，则对其进行索引。如果$consecutive为真，则数组键必须是从0开始的连续序列。

注意，空数组将被认为是索引的。

//

///2/ 摘要

public static function isIndexed($array, $consecutive = false): ?bool

//

///3/ 参数说明

$array:array；被检查的数组。

$consecutive:boolean；数组键是否必须是连续序列，以便将数组视为索引。

//

///4/ 返回值

boolean（数组是否是关联的）

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::isIndexed($arr);//true

$arr = ['a', 'b', 5 => 'c'];

$rs = ArrayHelper::isIndexed($arr);//true

$arr = ['a', 'b', 'key' => 'c'];

$rs = ArrayHelper::isIndexed($arr);//false

$arr = ['a', 'b', 5 => 'c'];

$rs = ArrayHelper::isIndexed($arr, true);//false

?>

//

(15)、isIn

///1/ 功能

检查数组或[[\Traversable]]是否包含元素。此方法与PHP函数[in\_array()]（http://php.net/manual/en/function.in-array.php）相同，但也适用于实现[[\Traversable]]接口的对象。

//

///2/ 概要

public static function isIn($needle, $haystack, $strict = false): bool

//

///3/ 参数说明

$needle:mixed；待查找的值。

$haystack:array|Traversable；要搜索的值集。

$strict:boolean；是否启用strict（`===`）比较。

//

///4/ 返回值

boolean（如果$needle在$haystack中找到，则为true，否则为false。）

//

///5/ 异常

如果$haystack既不可遍历也不是数组，则返回InvalidArgumentException。

//

///4/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::isIn('b', $arr);//true

$rs2 = ArrayHelper::isIn('d', $arr);//false

?>

//

(16)、isTraversable

///1/ 功能

检查变量是数组还是[[\Traversable]]。

此方法与PHP函数[is\_array()]（http://php.net/manual/en/function.is-array.php）相同，但另外适用于实现[[\Traversable]]接口的对象。

//

///1/ 概要

public static function isTraversable($var): bool

//

///2/ 参数说明

$var:mixed；正在评估的变量。

//

///3/ 返回值

boolean（$var是否类似于数组）

//

///4/ 示例

<?php

$arr = ['a'];

$rs = ArrayHelper::isTraversable($arr);//true

$arr2 = 'a';

$rs2 = ArrayHelper::isTraversable($arr);//false

?>

//

(17)、isSubset

///1/ 功能

检查数组或[[\Traversable]]是否是另一个数组的子集或[[\Traversable]]。

如果$needle的所有元素都包含在$ haystack中，则此方法将返回true。如果缺少至少一个元素，则返回false。

//

///2/ 概要

public static function isSubset($needles, $haystack, $strict = false): ?bool

//

///3/ 参数说明

$needles:array|Traversable；必须\*\*全部\*\*的值在$haystack中。

$haystack:array|Traversable；要搜索的值集。

$strict:boolean；是否启用strict（`===`）比较。

//

///4/ 返回值

boolean（true如果$needle是$haystack的子集，否则为false。）

//

///5/ 异常

如果$haystack或$needle既不可遍历也不是数组，则返回InvalidArgumentException。

//

///6/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::isSubset(['b', 'c'], $arr);//true

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::isSubset(['b', 'd'], $arr);//false

?>

//

(18)、filter

///1/ 功能

根据指定的规则过滤数组。

//

///2/ 概要

public static function filter($array, $filters): array

//

///3/ 参数说明

$array:array；原数组。

$filters:array；定义应从结果中保留或删除的数组键的规则。每条规则是：var = $array['var']将留在结果中；var.key = 只有$array['var']['key']将留在结果中；!var.key =$array['var']['key']将从结果中删除。

//

///4/ 返回值

array（过滤的数组）

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'description' => [

        'name' => 'swoft',

        'version' => '2.0'

    ]

];

$rs = ArrayHelper::filter($arr, ['id', 'description.version']);//['id'=>1,'description'=>['version'=>'2.0']]

?>

//

(19)、accessible

///1/ 功能

确定给定值是否可访问数组。

//

///2/ 概要

public static function accessible($value): bool

//

///3/ 参数说明

$value:mixed；

//

///4/ 返回值

boolean

//

///5/ 示例

<?php

$arr = ['a'];

$rs = ArrayHelper::accessible($arr);//true

$arr = 'a';

$rs = ArrayHelper::accessible($arr);//false

?>

//

(20)、exists

///1/ 功能

确定提供的数组中是否存在给定的键。

//

///2/ 概要

public static function exists($array, $key): bool

//

///3/ 参数说明

$array:ArrayAccess|array；

$key:string|int；

//

///4/ 返回值

boolean

//

//// 示例

<?php

$arr = ['id' => 1, 'name' => 'swoft'];

$rs = ArrayHelper::exists($arr, 'name');//true

$rs2 = ArrayHelper::exists($arr, 'description');//false

?>

//

(21)、get

///1/ 功能

使用“点”表示法从数组中获取项目。

//

///2/ 概要

public static function get($array, $key = null, $default = null)

//

///3/ 参数说明

$array:ArrayAccess|array；

$key:string；

$default:mixed；

//

///4/ 返回值

mixed

//

///5/ 示例

<?php

$arr = ['id' => 1, 'name' => 'swoft'];

$rs = ArrayHelper::get($arr, 'name');//swoft

$rs2 = ArrayHelper::get($arr, 'description');//null

$rs3 = ArrayHelper::get($arr, 'description', '2.0');//'2.0'

?>

//

(22)、has

///1/ 功能

使用“点”表示法检查数组中是否存在项目。

//

///2/ 概要

public static function has($array, $key): bool

//

///3/ 参数说明

$array:ArrayAccess|array；

$key:string；

//

///4/ 返回值

boolean

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'description' => [

        'name' => 'swoft',

        'version' => '2.0'

    ]

];

$rs = ArrayHelper::has($arr, 'id');//true

$rs2 = ArrayHelper::has($arr, 'name');//false

$rs3 = ArrayHelper::has($arr, 'description.name');//true

?>

//

(23)、set

///1/ 功能

使用“点”表示法将数组项设置为给定值。如果没有为该方法指定key，则将替换整个数组。

//

///2/ 概要

public static function set(&$array, $key, $value): array

//

///3/ 参数说明

$array:array；

$key:string；

$value:mixed；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['id' => 1];

$rs = ArrayHelper::set($arr, 'name', 'swoft');//['id'=>1,'name'=>'swoft']

?>

//

(24)、insert

///1/ 功能

将一个数组插入另一个数组。

//

///2/ 概要

public static function insert(array &$array, int $index, ...$insert): void

//

///3/ 参数说明

$array:array；

$index:int；

$insert:array；

//

///4/ 返回值

void

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c'];

ArrayHelper::insert($arr, 2, 'd', 'e');//['a','b','d','e','c']

?>

//

(25)、wrap

///1/ 功能

如果给定的值不是数组而不是null，则将其包装在一个中。

//

///2/ 概要

public static function wrap($value): array

//

///3/ 参数说明

$value:mixed；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ArrayHelper::wrap(['a']);//['a']

$arr2 = ArrayHelper::wrap('a');//['a']

?>

//

(26)、isArrayable

///1/ 功能

判断是否是数组或是否是Arrayable的对象。

//

///2/ 概要

public static function isArrayable($value): bool

//

///3/ 参数说明

$value:mixed；

//

///4/ 返回值

boolean

//

///5/ 示例

<?php

$arr = ['a'];

$rs = ArrayHelper::isArrayable($arr);//true

$arr = 'a';

$rs = ArrayHelper::isArrayable($arr);//false

?>

//

(27)、flatten

///1/ 功能

将多维数组展平为单个级别。

//

///2/ 概要

public static function flatten(array $array, int $depth = PHP\_INT\_MAX): array

//

///3/ 参数说明

$array:array；

$depth:int；

//

///4 返回值

array

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'description' => [

        'name' => 'swoft',

        'version' => '2.0'

    ]

];

$rs = ArrayHelper::flatten($arr);//[1,'swoft','2.0']

?>

//

(28)、findSimilar

///1/ 功能

从 array|Iterator中查找类似的文本。

//

///2/ 概要

public static function findSimilar(string $need, $iterator, int $similarPercent = 45): array

//

///3/ 参数说明

$need:string；

$iterator:Iterator|array；

$similarPercent:int；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = [

    'swoft',

    'swoft-2',

    'yii',

    'thinkphp',

    'test-swoft-cloud'

];

$rs = ArrayHelper::findSimilar('swoft', $arr);//['swoft','swoft-2','test-swoft-cloud']

?>

//

(29)、getKeyMaxWidth

///1/ 功能

获取关键最大宽度。

//

///2/ 概要

public static function getKeyMaxWidth(array $data, bool $expectInt = false): int

//

///3/ 参数说明

$data:array；['key1'=>'value1','key2-test'=>'vaule2']；

$expectInt:bool；

//

///4/ 返回值

int

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'name' => 'swoft',

    'version' => '2.0',

    'description' => 'php framework'

];

$rs = ArrayHelper::getKeyMaxWidth($arr);//11

?>

//

(30)、first

///1/ 功能

返回通过给定真值测试的数组中的第一个元素。

//

///2/ 概要

public static function first($array, callable $callback = null, $default = null)

//

///3/ 参数说明

$array:array；

$callback:callable|null；

$default:mixed；

//

///4/返回值

mixed

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c', 'd'];

$rs = ArrayHelper::first($arr, function ($value) {

    return $value == 'c';

});//c

?>

//

(31)、where

///1/ 功能

使用给定的回调过滤数组。

//

///2/ 概要

public static function where($array, callable $callback)

//

///3/ 参数说明

$array:array；

$callback:callable；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c', 'd'];

$rs = ArrayHelper::where($arr, function ($value) {

    return $value == 'c';

});//[2=>'c']

?>

//

(32)、query

///1/ 功能

将数组转换为查询字符串。

//

///2/ 概要

public static function query(array $array): string

//

///3/ 参数说明

$array:array；

//

///4/ 返回值

string

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'name' => 'swoft',

];

$rs = ArrayHelper::query($arr);//'id=1&name=swoft'

?>

//

(33)、only

///1/ 功能

从给定数组中获取项目的子集。

//

///2/ 概要

public static function only(array $array, array $keys): array

//

///3/ 数说明

$array:array；

$keys:array；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = [

    'id' => 1,

    'name' => 'swoft',

];

$rs = ArrayHelper::only($arr, ['name']);//['name'=>'swoft']

?>

//

(34)、last

///1/ 功能

返回通过给定真值测试的数组中的最后一个元素。

//

///2/ 概要

public static function last($array, callable $callback = null, $default = null)

//

///3/ 参数说明

$array:array；

$callback:callable|null；

$default:mixed；

//

///4/ 返回值

mixed

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c', 'd'];

$rs = ArrayHelper::last($arr, function ($value) {

    return $value == 'c';

});//'c'

?>

//

(35)、pluck

///1/ 功能

从数组中获取值的数组。

//

///2/ 概要

public static function pluck($array, $value, $key = null)

//

///3/ 参数说明

$array:array；

$value:string|array；

$key:string|array|null；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = [

    ['id' => 1, 'name' => 'swoft'],

    ['id' => 2, 'name' => 'yii'],

];

$rs = ArrayHelper::pluck($arr, 'name');//['swoft','yii']

?>

//

(36)、collapse

///1/ 功能

将数组的数组折叠成单个数组。

//

///2/ 概要

public static function collapse($array)

//

///3/ 参数说明

$array:array；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = [

    ['id' => 1, 'name' => 'swoft', 'version' => '2.0'],

    ['id' => 2, 'name' => 'yii'],

];

$rs = ArrayHelper::collapse($arr);//['id'=>2,'name'=>'yii','version'=>'2.0']

?>

//

(37)、crossJoin

///1/ 功能

交叉连接给定的数组，返回所有可能的排列。

//

///2/ 概要

public static function crossJoin(...$arrays)

//

///3/ 参数说明

$array:array；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['a'];

$arr2 = ['b'];

$rs = ArrayHelper::crossJoin($arr, $arr2);//['a','b']

?>

//

(38)、prepend

///1/ 功能

将一个项推到数组的开头。

//

///2/ 概要

public static function prepend($array, $value, $key = null)

//

///3/ 参数说明

$array:array；

$value:mixed；

$key:mixed；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::prepend($arr, 'd');//['d','a','b','c']

?>

//

(39)、random

///1/ 功能

从数组中获取一个或指定数量的随机值。

//

///2/ 概要

public static function random($array, $number = null)

//

///3/ 参数说明

$array:array；

$number:int|null；

//

///4/ 返回值

mixed

//

///5/ 异常

InvalidArgumentException

//

///6/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::random($arr);

in\_array($rs,$arr);//true

?>

//

(40)、shuffle

///1/ 功能

对给定数组进行洗牌并返回结果。

//

///2/ 概要

public static function shuffle($array, $seed = null)

//

///3/ 参数说明

$array:array；

$seed:int|null；

//

///4/ 返回值

array

//

///5/ 示例

<?php

$arr = ['a', 'b', 'c'];

$rs = ArrayHelper::shuffle($arr);

var\_dump(count($arr) === count($rs));//true

?>

--

--

第四篇 核心

--

--

一、命令行

( http://swoft.org/docs/2.x/zh-CN/console/index.html )

--

0、概述

swoft 提供功能强大的命令行应用处理功能。swoft 的 http server, websocket server 等都是通过命令行启动和管理。

(图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/console/cli-app.jpg)



//

(1)、安装

composer require swoft/console

//

(2)、功能特性

命令行应用, 命令行的解析运行，基于注解 @Command @CommandMapping 自动收集注册。

支持给命令、命令组设置别名,一个命令可以有多个别名。

功能全面的命令行的选项参数解析(命名参数，短选项，长选项 ...)，命令行下 input, output 管理、使用。

命令方法注释自动解析为帮助信息（支持 @CommandOption @CommandArgument @example 等注解）。

支持输出多种颜色风格的消息文本(info, comment, success, warning, danger, error ... )。

常用的特殊格式信息显示(section, panel, padding, helpPanel, table, title, list, multiList)。

丰富的动态信息显示(pending/loading, pointing, spinner, counterTxt, dynamicText, progressTxt, progressBar)。

常用的用户信息交互支持(select, multiSelect, confirm, ask/question, askPassword/askHiddenInput)。

颜色输出是 windows linux mac 兼容的，不支持颜色的环境会自动去除相关CODE。

支持协程和非协程（传统同步阻塞）两种方式运行命令。

内置Phar打包工具类，可以方便的将应用打包成phar文件，方便作为工具分发和使用。

快速的为当前应用生成 bash/zsh 环境下的自动补全脚本(TODO)。

说明：swoft/console( https://github.com/inhere/php-console ) 基于 inhere/php-console( https://github.com/symfony/console ) 改进，并参考了 symfony/console 部分特性。

//

(3)、Git仓库

Github https://github.com/swoft-cloud/swoft-console

//

(4)、参与贡献

欢迎参与贡献，您可以进行如下操作。

fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )。

修改代码然后发起 PR。

关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、命令行配置

( http://swoft.org/docs/2.x/zh-CN/console/config.html )

(1)、AutoLoader配置

<?php declare(strict\_types=1);

namespace Swoft\Console;

use Swoft\Console\Router\Router;

use Swoft\Helper\ComposerJSON;

use Swoft\SwoftComponent;

use function dirname;

/\*\*

 \* class AutoLoader

 \* @since 2.0

 \*/

final class AutoLoader extends SwoftComponent

{

    /\*\*

     \* @return bool

     \*/

    public function enable(): bool

    {

        return true;

    }

    /\*\*

     \* Get namespace and dirs

     \*

     \* @return array

     \*/

    public function getPrefixDirs(): array

    {

        return [

            \_\_NAMESPACE\_\_ => \_\_DIR\_\_,

        ];

    }

    /\*\*

     \* Metadata information for the component

     \*

     \* @return array

     \*/

    public function metadata(): array

    {

        $jsonFile = dirname(\_\_DIR\_\_) . '/composer.json';

        return ComposerJSON::open($jsonFile)->getMetadata();

    }

    /\*\*

     \* {@inheritDoc}

     \*/

    public function beans(): array

    {

        return [

            'cliApp'    => [

                'class' => Application::class,

            ],

            'cliRouter' => [

                'class' => Router::class,

            ],

            'cliDispatcher' => [

                'class' => ConsoleDispatcher::class,

            ],

        ];

    }

}

?>

--

2、定义命令

( http://swoft.org/docs/2.x/zh-CN/console/definition.html )

一个命令由命令组和执行命令组成，一个类就是一个命令组，类里面的方法对应操作命令，一个命令的运行，是通过执行命令组对应的操作命令。命令逻辑里面可以使用 Swoft 所有功能。

//

(1)、命令结构

                                        value of option: opt1

                                option: opt1  |

                                       |      |

php bin/swoft group:cmd john male 43 --opt1 value1 -y

        |         |      |    |   |                 |

     script    command   |    |   |\_\_\_\_\_        option: yes, it use shortcat: y, and it is a bool, so no value.

                         |    |\_\_\_     |

                 argument: name  |   argument: age

                            argument: sex

//

///1/ 参数与选项

|1>、没有 - 开头的都认为是参数 (eg: status=2 arg0)。

|2>、反之，以 - 开头的则是选项数据：-- 开头的是长选项(long-option)；一个 - 开头的是短选项(short-option)。支持混合式选项的赋值 --id=154 和 --id 154 是等效的。

注意: 输入如下的字符串将会认为是布尔值，on|yes|true -- true，off|no|false -- false。

//

(2)、注解

命令的定义主要通过 @Command 和 @CommandMapping两个注解。 @Command 定义命令组名称，@CommandMapping 定义操作命令的映射关系。命令的命令使用帮助信息，也是通过注解完成定义。

///1/ Command

@Command 定义命令组，标记一个类为console命令类。作用域：class。

// 拥有属性

name string 定义命令组名称，如果缺省，根据类名称自动解析。

alias string 命令组别名，通过别名仍然可以访问它。允许多个，以逗号隔开即可。

desc string 命令组描述信息说明，支持颜色标签。

coroutine bool 定义是否为协程下运行，默认 true, 框架会启动一个协程运行此组里面的命令。

Tips: 若 desc 为空，将会自动解析类的第一行注释作为命令组描述。

//

///2/ CommandMapping

@CommandMapping 定义操作命令映射关系，标明了一个具体的命令。作用域：method。

// 拥有属性

name string 定义命令组名称，如果缺省，会执行使用方法名称。

alias string 命令别名，通过别名仍然可以访问它。允许多个，以逗号隔开即可。

desc string 命令的描述信息说明，支持颜色标签。

Tips: 若 desc 为空，将会自动解析类的第一行注释作为描述。

//

///3/ CommandOption

@CommandOption 定义一个命令的选项。作用域：method|class。

// 拥有属性

name string 必填项 定义命令选项名称。eg: opt。

short string 定义命令选项名称的短选项。

default mixed 命令选项的默认值。

desc string 命令选项的描述信息说明，支持颜色标签。

type string 命令选项的值类型。

mode int 命令选项的值输入限定：必须，可选 等。

Tips: 特别的 @CommandOption 可以用 command 类注释上面，这样子相当于给里面所有的命令都加了公共选项。

／／

///4/ CommandArgument

@CommandArgument 定义一个命令的参数。作用域：method。

// 拥有属性

name string 必填项 定义命令参数名称。eg: opt。

default mixed 命令参数的默认值。

desc string 命令参数的描述信息说明，支持颜色标签。

type string 命令参数的值类型。

mode int 命令参数的值输入限定：必须，可选 等。

Tips: 命令参数是根据输入位置(有顺序的)来获取的，name 是代码里给这个位置的参数添加的命名。

//

(3)、example 注释

@example 注释会被特殊处理(不是注解)，如果你的命令方法上面有这个注释，它的内容也会被显示到命令帮助信息上面。

//

(4)、代码示例

下面是 swoftcli 里的 serve:run 命令定义。命令帮助显示效果如下所示( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/hot-restart-help.jpg )。

/\*\*

 \* Provide some commands for manage and watch swoft server project

 \*

 \* @Command()

 \*/

class ServeCommand

{

    /\*\*

     \* Start the swoft server and monitor the file changes to restart the server

     \*

     \* @CommandMapping()

     \* @CommandArgument("targetPath", type="path",

     \*     desc="Your swoft project path, default is current work directory"

     \* )

     \* @CommandOption("interval", type="integer", default=3,

     \*     desc="Interval time for watch files, unit is seconds"

     \* )

     \* @CommandOption(

     \*     "bin-file", short="b", type="string", default="bin/swoft",

     \*     desc="Entry file for the swoft project"

     \* )

     \* @CommandOption(

     \*     "start-cmd", short="c", type="string", default="http:start",

     \*     desc="the server startup command to be executed"

     \* )

     \* @CommandOption(

     \*     "watch", short="w", default="app,config", type="directories",

     \*     desc="List of directories you want to watch, relative the <cyan>targetPath</cyan>"

     \* )

     \* @example

     \*   {binFile} run -c ws:start -b bin/swoft /path/to/php/swoft

     \* @param Input $input

     \*/

    public function run(Input $input): void

    {

        Show::aList([

            'options'   => $input->getOpts(),

            'arguments' => $input->getArgs(),

        ]);

    }

}

//

(5)、使用运行

现在你可以执行 php bin/swoft, 命令列表中将会显示 serve 组命令。

执行 php bin/swoft serve 或者 php bin/swoft serve -h 将会看到 serve组里拥有的具体命令。

执行 php bin/swoft serve:run -h 将会看到此命令的完整帮助信息。

--

3、命令运行

( http://swoft.org/docs/2.x/zh-CN/console/usage.html )

完成定义命令后，可以执行命令，处理对应业务逻辑，如果不特殊设置，swoft 默认在协程环境运行命令。

//

(1)、特别说明

command 里是始终不可能直接操作server的。 你每运行一次 command，都是在一个全新的进程里，除了使用一样的代码，其他毫无关系。

//

(2)、查看命令

//// 查看当前已经定义的所有命令组

php bin/swoft

// 或者

php bin/swoft -h

php bin/swoft --help

( 效果图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/console/cli-app.jpg )



//

(3)、查看版本

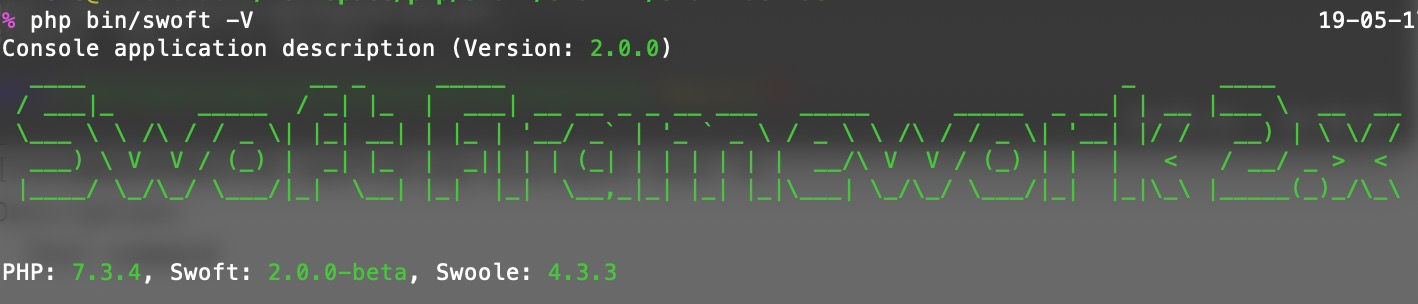
//// 查看当前 swoft 框架版本信息

php bin/swoft -V

// 或者

php bin/swoft --version

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/console/cli-version.jpg )



//

(4)、命令帮助信息

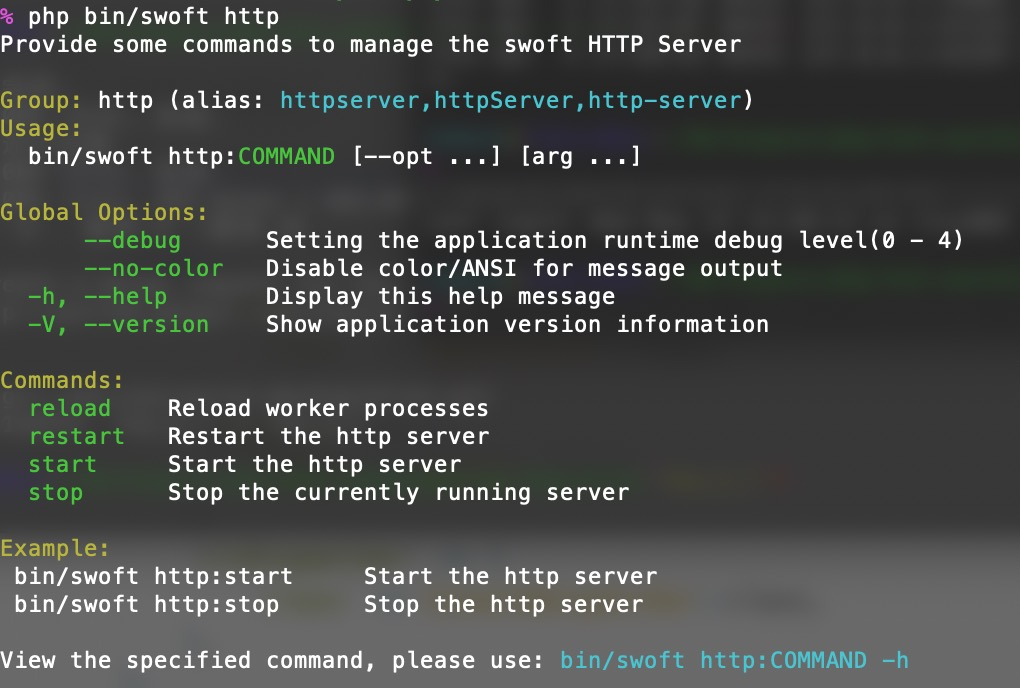
///1/ 命令组帮助信息

php bin/swoft http

// 或者

php bin/swoft http -h

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/console/cli-group.jpg )



//

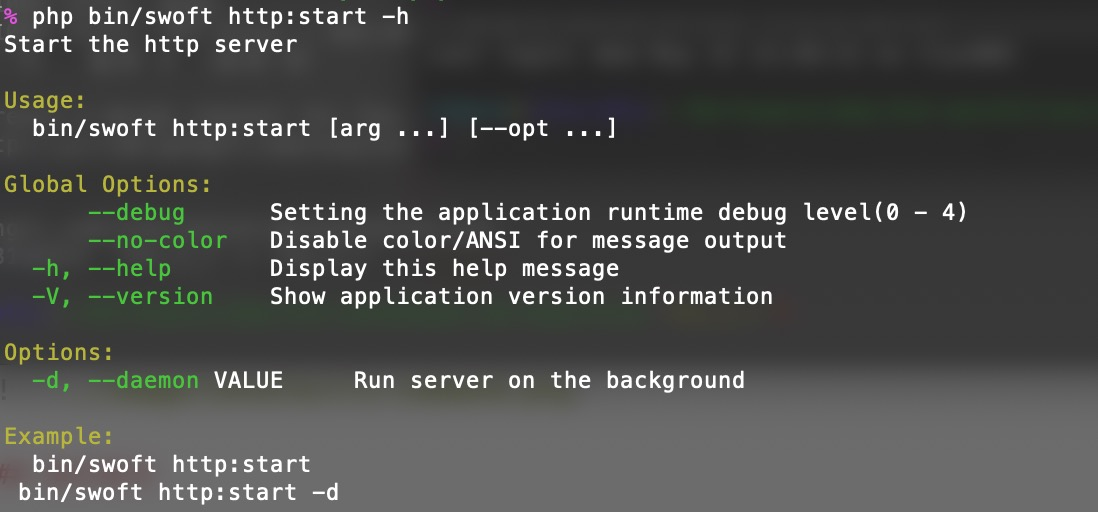
///2/ 具体命令帮助信息

php bin/swoft http:start -h

// 或者

php bin/swoft http:start --help

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/console/cli-command.jpg )



//

(5)、执行命令

// 运行命令组下面，某个操作命令

php bin/swoft http:start

--

4、输入对象

( http://swoft.org/docs/2.x/zh-CN/console/input.html )

输入对象是 Swoft\Console\Input\Input 的实例，用于获取用户输入的命令参数选项等信息。 命令逻辑里面，可以通过函数参数和全局函数获取输入输出对象。

//

(1)、获取输入对象

///1/ 通过方法参数

如果需要使用输入和输出对象，可以在操作命令函数上，定义输入和输出对象，底层框架会自动注入对象。

/\*\*

 \* Test command

 \*

 \* @Command(coroutine=true)

 \*/

class TestCommand

{

    /\*\*

     \* @param Input  $input

     \* @param Output $output

     \*

     \* @CommandMapping("test2")

     \*/

    public function test(Input $input, Output $output)

    {

        // ......

    }

}

//

///2/ 使用全局函数

/\*\*

 \* Test command

 \*

 \* @Command(coroutine=true)

 \*/

class TestCommand

{

    /\*\*

     \* @CommandMapping()

     \*/

    public function demo()

    {

        $input = \input();

        $output = \output();

        // ......

    }

}

//

(2)、开始使用

通过前一篇 定义命令( http://swoft.org/docs/2.x/zh-CN/console/definition.html ) 文章，我们已经知道了什么是命令参数，命令选项。现在，终端中执行如下命令，用于演示参数选项等信息的解析。

$ php bin/swoft demo:test status=2 name=john arg0 -s=test --page 23 --id=154 -e dev -v vvv -d -rf --debug --test=false

注意: 输入如下的字符串将会认为是布尔值，on|yes|true -- true；off|no|false -- false。

//

(3)、获取基本信息

echo $input->getScript();  // 'bin/swoft' 执行的入口脚本文件

echo $input->getCommand(); // 'http:start' 命令名称 解析到的第一个参数将会被认为是命令名称，并且不会再存入到 参数列表中

echo $input->getPwd(); // 当前工作目录

//

(4)、命令参数信息

通常的参数如 arg0 只能根据 index key 来获取值。但是提供以等号(=)连接的方式来指定参数名(eg: status=2)。

// 打印所有的参数信息

var\_dump($input->getArgs());

// output

array(3) {

  'status' => string(1) "2"

  'name' => string(4) "john"

  [0] => string(4) "arg0"

}

///1/ 获取命令参数值

// argument

$first = $input->getFirstArg(); // 'arg0'

$status = $input->getArg('status', 'default value'); // '2'

$status = $input->getInt('status'); // 2

// 获取一个必须的参数，若用户没有输入值，将会抛出错误信息

$id = $input->getRequiredArg('id');

//

(5)、命令选项信息

// 获取解析后的选项信息

没有值的选项，将设置默认值为 bool(true)。

短选项不仅仅只是以一个 - 开头，而且名称 只能是一个字符。

多个(默认值的)短选项可以合并到一起写。如 -rf 会被解析为两个短选项 'r' => bool(true) 'f' => bool(true)。

//

// 打印所有的选项信息

var\_dump($input->getOpts());

// var\_dump($input->getLOpts()); // 只打印长选项信息

// var\_dump($input->getSOpts()); // 只打印短选项信息

// output

array(10) {

  's' => string(4) "test"

  'e' => string(3) "dev"

  'v' => string(3) "vvv"

  'd' => bool(true)

  'r' => bool(true)

  'f' => bool(true)

  'page' => string(2) "23"

  'id' =>   string(3) "154"

  'debug' => bool(true)

  'test' => bool(false)

}

//

///1/ 获取选项值

// 输入对象中提供了非常多的选项值获取方法，方便快速的获取需要的信息。

// option

$page = $input->getOpt('page') // '23'

$page = $input->getIntOpt('page') // 23

$debug = $input->getBoolOpt('debug') // True

$test = $input->getBoolOpt('test') // False

$d = $input->getBoolOpt('d') // True

// 获取到一个值就返回，对同一个含义的选项选项非常有用

$showHelp = $input->sameOpt(['h','help']);

// 获取一个必须的选项，若用户没有输入值，将会抛出错误信息

$id = $input->getRequiredOpt('id');

//

(6)、读取用户输入

echo "Your name:";

$name = $input->read();

echo 'input is ' . $name; // 'inhere'

// 效果(in terminal)

$ Your name: inhere

$ input is inhere

也可以直接将消息文本放入参数 $name = $input->read("Your name:");

--

5、输出对象

( http://swoft.org/docs/2.x/zh-CN/console/output.html )

(1)、获取输出对象

输出对象是 Swoft\Console\Output\Output 的实例，用于显示信息到控制台。命令逻辑里面，可以通过函数参数和全局函数获取输入输出对象。

//

///1/ 通过方法参数

如果需要使用输入和输出对象，可以在操作命令函数上，定义输入和输出对象，底层框架会自动注入对象。

/\*\*

 \* Test command

 \*

 \* @Command(coroutine=true)

 \*/

class TestCommand

{

    /\*\*

     \* @param Input  $input

     \* @param Output $output

     \*

     \* @CommandMapping("test2")

     \*/

    public function test(Input $input, Output $output)

    {

        // ......

    }

}

//

///2/ 使用全局函数

/\*\*

 \* Test command

 \*

 \* @Command(coroutine=true)

 \*/

class TestCommand

{

    /\*\*

     \* @CommandMapping()

     \*/

    public function demo()

    {

        $input = \input();

        $output = \output();

        // ......

    }

}

--

6、数据展示

( http://swoft.org/docs/2.x/zh-CN/console/data-show.html )

Console数据展示 - 提供格式化信息的输出显示。主要功能封装在命名空间 Swoft\Console\Advanced\Formatter 下，提供了 Swoft\Console\Helper\Show 辅助类来快速使用它们。

//

(1)、标题文本输出

// 使用 Show::title()/$output->title()

public static function title(string $title, array $opts = [])

//

(2)、段落式文本输出

// 使用 Show::section()/$output->section()

public static function section(string $title, string|array $body, arr

//

(3)、列表数据展示输出

public static function aList(array $data, string $title, array $opts = [])

//

$data array 列表数据。可以是key-value 形式，也可以只有 value，还可以两种混合。

$title string 列表标题。可选的。

$opts array 选项设置(同表格、面板的选项)。

leftChar 左侧边框字符，默认两个空格，也可以是其他字符(eg: \* .)；keyStyle 当key-value 形式时，渲染 key 的颜色风格，默认 info, 设为空即是不加颜色渲染；titleStyle 标题的颜色风格，默认 comment。

//

提示：aList 的默认选项，可以渲染一个命令的帮助信息。

//

///1/ 使用 Show::aList()/$output->aList()

$title = 'list title';

$data = [

     'name'  => 'value text', // key-value

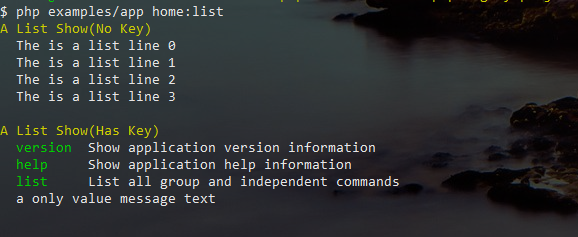
     'name2' => 'value text 2',

     'more info please XXX', // only value

];

Show::aList($data, $title);

( 渲染效果图 https://raw.githubusercontent.com/inhere/php-console/master/docs/screenshots/fmt-list.png )



//

(4)、多列表数据展示输出

public static function mList(array $data, array $opts = [])

mList 的默认选项，可以渲染一组命令的帮助信息。效果与 helpPanel() 相同，并且自定义性更高。

//

///1/ 使用 Show::mList()/$output->mList() 别名方法 Show::multiList()

$data = [

  'list1 title' => [

     'name' => 'value text',

     'name2' => 'value text 2',

  ],

  'list2 title' => [

     'name' => 'value text',

     'name2' => 'value text 2',

  ],

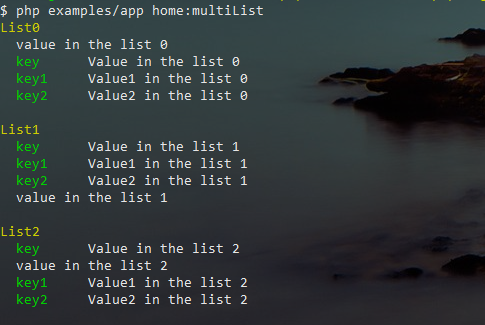
  // ... ...

];

Show::mList($data);

//

( 渲染效果 https://raw.githubusercontent.com/inhere/php-console/master/docs/screenshots/fmt-multi-list.png )



//

(5)、面板展示信息输出

public static function panel(mixed $data, $title = 'Information Panel', $borderChar = '\*')

展示信息面板。比如 命令行应用 开始运行时需要显示一些 版本信息，环境信息等等。

//

///1/ 使用 Show::panel()/$output->panel()

$data = [

    'application version' => '1.2.0',

    'system version' => '5.2.3',

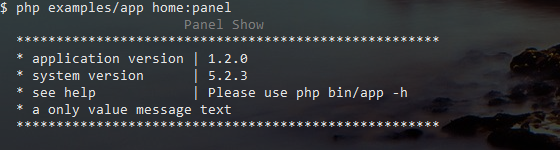
    'see help' => 'please use php bin/app -h',

    'a only value message',

];

Show::panel($data, 'panel show', '#');

( 渲染效果 https://raw.githubusercontent.com/inhere/php-console/master/docs/screenshots/fmt-panel.png )



//

(6)、数据表格信息输出

public static function table(array $data, $title = 'Data Table', array $opts = [])

//

///1/ 使用 Show::table()/$output->table()

/// 可直接渲染从数据库拉取的数据(会自动提取字段名作为表头)

// like from database query's data.

$data = [

 [ col1 => value1, col2 => value2, col3 => value3, ... ], // first row

 [ col1 => value4, col2 => value5, col3 => value6, ... ], // second row

 ... ...

];

Show::table($data, 'a table');

//

/// 自己构造数据时，还要写字段名就有些麻烦了。可以通过选项配置 $opts 手动配置表头字段列表。

// use custom head

$data = [

 [ value1, value2, value3, ... ], // first row

 [ value4, value5, value6, ... ], // second row

 // ... ...

];

$opts = [

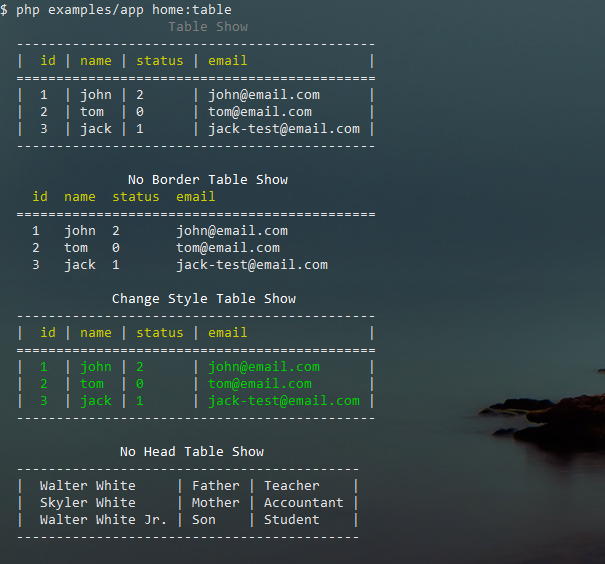
  'showBorder' => true,

  'columns' => [col1, col2, col3, ...]

];

Show::table($data, 'a table', $opts);

( 渲染效果预览图 https://raw.githubusercontent.com/inhere/php-console/master/docs/screenshots/table-show.png )



//

(7)、渲染帮助信息面板

public static function helpPanel(array $config, $showAfterQuit = true)

//

///1/ 使用 Show::helpPanel()/$output->helpPanel()

Show::helpPanel([

    Show::HELP\_DES => 'a help panel description text. (help panel show)',

    Show::HELP\_USAGE => 'a usage text',

    Show::HELP\_ARGUMENTS => [

        'arg1' => 'arg1 description',

        'arg2' => 'arg2 description',

    ],

    Show::HELP\_OPTIONS => [

        '--opt1' => 'a long option',

        '-s' => 'a short option',

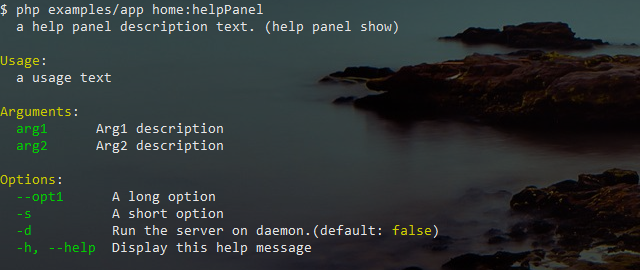
        '-d' => 'Run the server on daemon.(default: <comment>false</comment>)',

        '-h, --help' => 'Display this help message'

    ],

], false);

( 渲染效果预览 https://raw.githubusercontent.com/inhere/php-console/master/docs/screenshots/fmt-help-panel.png )



--

--

二、Http Server

( http://swoft.org/docs/2.x/zh-CN/http-server/index.html )

--

0、概述 (http 服务器)

( http://swoft.org/docs/2.x/zh-CN/http-server/index.html )

基于 \Swoole\Http\Server 实现的协程HTTP服务, 框架层做了很好的封装, 用户按照传统的 MVC 方式编写代码, 就能获得协程带来的超高性能.

//

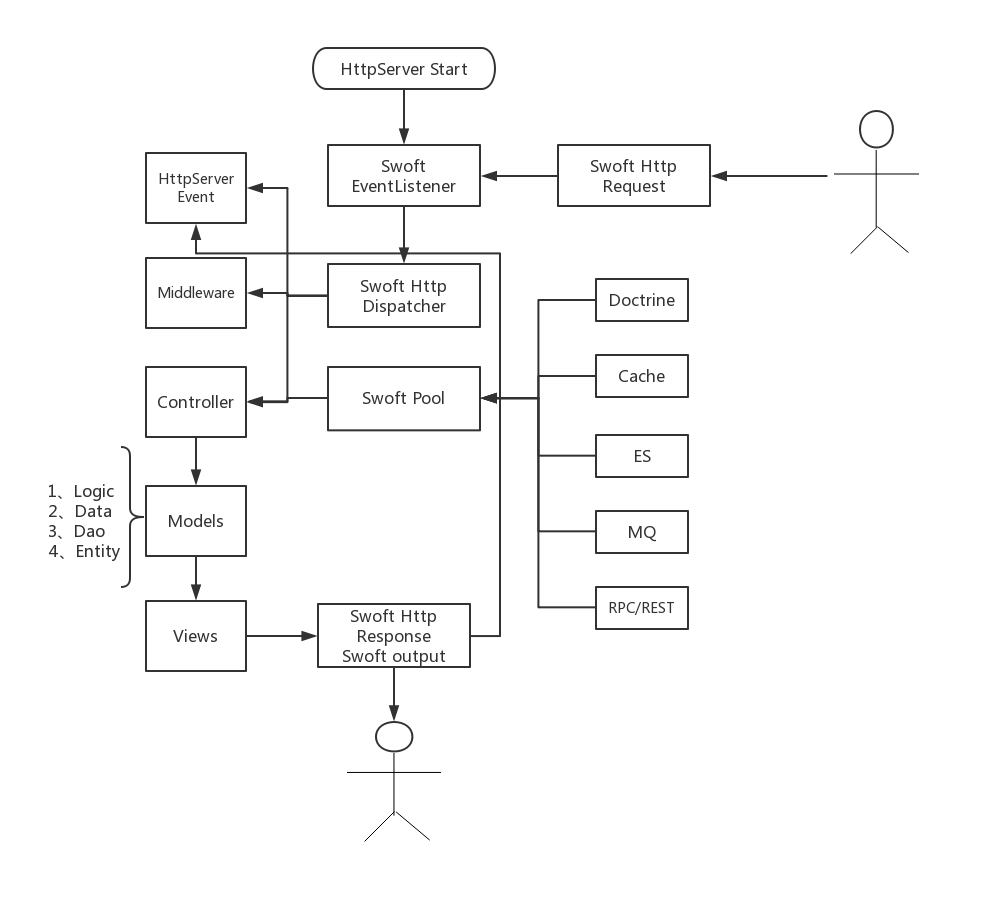
(1)、安装

omposer require swoft/http-server

//

(2)、http 生命周期

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/http-server/process/1.png )

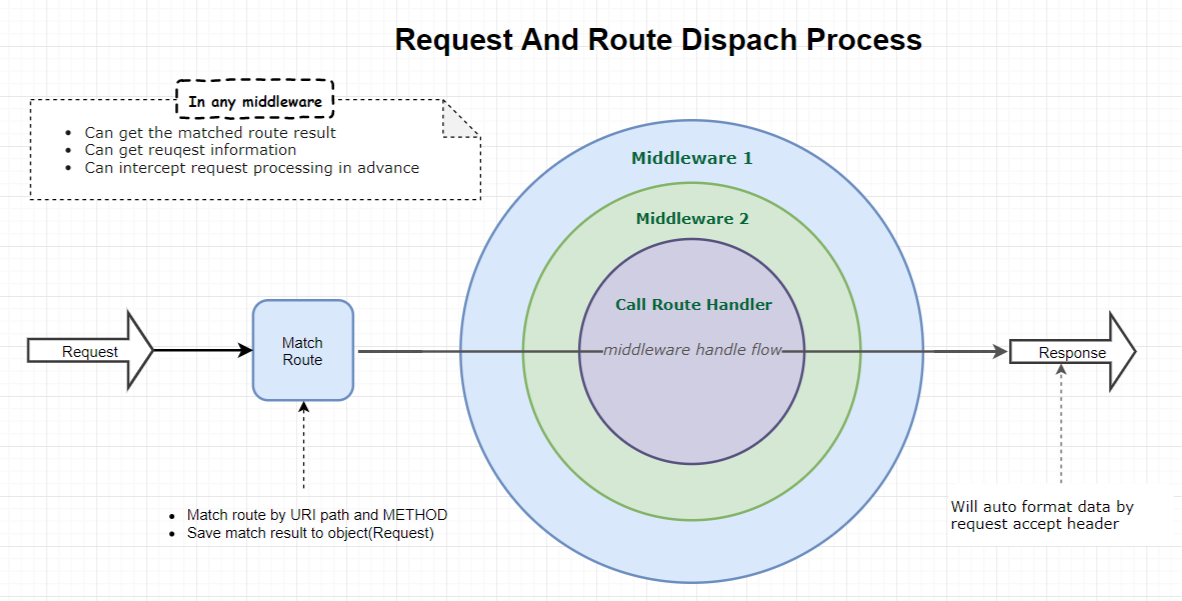


了解请求生命周期, 有利于理解HTTP服务各组件, 编写出更好代码.

//

(3)、请求处理流程

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/http-server/swoft2-request-flow.png )



//

(4)、功能特色

基于 PSR-7 的 HTTP 消息实现。

基于 PSR-15 的中间件。

@Controller 灵活的控制器注解。

@RequestMapping 灵活的路由注解。

//

(5)、Git仓库

Github https://github.com/swoft-cloud/swoft-http-server

//

(6)、参与贡献

欢迎参与贡献，您可以：fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )；修改代码然后发起 PR；关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、常用命令

( http://swoft.org/docs/2.x/zh-CN/http-server/command.html )

(0)、Http服务命令

///1/ 在项目根目录执行如下命令

$ php bin/swoft http

Provide some commands to manage the swoft HTTP Server

Group: http (alias: httpserver,httpServer,http-server)

Usage:

  bin/swoft http:COMMAND [--opt ...] [arg ...]

Global Options:

      --debug      Setting the application runtime debug level(0 - 4)

      --no-color   Disable color/ANSI for message output

  -h, --help       Display this help message

  -V, --version    Show application version information

Commands:

  reload    Reload worker processes

  restart   Restart the http server

  start     Start the http server

  stop      Stop the currently running server

Example:

 bin/swoft http:start     Start the http server

 bin/swoft http:stop      Stop the http server

View the specified command, please use: bin/swoft http:COMMAND -h

//

Http的命令都在 Commands。

reload 重新加载 worker 进程。

restart 重启 Http 服务器。

start 启动 Http 服务器。

stop 停止 Http 服务器。

//

(1)、使用

///1/ 前台运行

$ php bin/swoft http:start

                            Information Panel

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* HTTP     | Listen: 0.0.0.0:18306, type: TCP, mode: Process, worker: 1

  \* rpc      | Listen: 0.0.0.0:18307, type: TCP

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HTTP server start success !

//

///2/ 后台运行

$ php bin/swoft http:start -d

                            Information Panel

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* HTTP     | Listen: 0.0.0.0:18306, type: TCP, mode: Process, worker: 1

  \* rpc      | Listen: 0.0.0.0:18307, type: TCP

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HTTP server start success !

//

在浏览器通过 http://127.0.0.1:18306/ 访问Http服务。

--

2、配置参数

( http://swoft.org/docs/2.x/zh-CN/http-server/setting.html )

//

(0)、Http Server 配置参数

在应用下的 app/bean.php 配置server，下面列举了一些简单的配置，你也可以自由组合同时提供多种服务。

//

(1)、可配置项

可配置项用于 http server bean 配置，除了 class 其他都是 http server 的属性。

class 指定 Http Server 的处理类。

port 指定 Http Server 的端口.

listener 指定其他一同启动的服务，添加端口服务监听，可以多个，rpc 启动 RPC 服务。

on 配置监听的事件，注册事件、设置对应事件的处理监听，事件触发组件调用，在任务里面使用。

setting 这里是参考 Swoole Server配置选项。

pidFile 设置进程 pid文件 位置，默认值 @runtime/swoft.pid。

mode 运行的模式，参考 Swoole Server 构造函数 第三个参数。

type 指定Socket的类型，支持TCP、UDP、TCP6、UDP6、UnixSocket Stream/Dgram 等 Swoole Server 构造函数 第四个参数。

//

(2)、基础配置

Swoft应用的 Http Server 配置在 app/bean.php 中。在这个文件里，你可以看到 Http Server数组里面包含了 Http Server 的基本信息。

'httpServer' => [

    'class'    => HttpServer::class,

    'port'     => 18306,

    /\* @see HttpServer::$setting \*/

    'setting'  => [

        'log\_file' => alias('@runtime/swoole.log'),

    ]

],

//

(3)、启用Task处理

///1/ 启用Task进程处理任务

'httpServer' => [

    'class'    => HttpServer::class,

    'port'     => 18306,

    'on'       => [

        // Enable task must task and finish event

        SwooleEvent::TASK   => \bean(TaskListener::class),

        SwooleEvent::FINISH => \bean(FinishListener::class)

    ],

    /\* @see HttpServer::$setting \*/

    'setting'  => [

        'task\_worker\_num'       => 12,

        'task\_enable\_coroutine' => true

    ]

],

//

(4)、启用RPC支持

'httpServer' => [

    'class'    => HttpServer::class,

    'port'     => 18306,

    'listener' => [

        'rpc' => \bean('rpcServer')

    ],

    /\* @see HttpServer::$setting \*/

    'setting'  => [

        'task\_worker\_num'       => 12,

        'task\_enable\_coroutine' => true

    ]

],

'rpcServer'  => [

    'class' => ServiceServer::class,

    'port' => 18308,

],

//

(5)、启用https支持

在swoft里启用 https 支持也非常简单，添加如下的配置即可。

'httpServer' => [

    'type' => SWOOLE\_SOCK\_TCP | SWOOLE\_SSL,

    /\* @see HttpServer::$setting \*/

    'setting'  => [

        'ssl\_cert\_file' => '/my/certs/2288803\_www.domain.com.pem',

        'ssl\_key\_file'  => '/my/certs/2288803\_www.domain.com.key',

    ]

]

注意： 你必须安装 OpenSSL 库，并且确保安装swoole时是启用了 ssl 选项的。同时，需要设置 'type' => SWOOLE\_SOCK\_TCP | SWOOLE\_SSL。

--

3、控制器

( http://swoft.org/docs/2.x/zh-CN/http-server/controller.html )

控制器作为HTTP服务的核心组件，串接起一次请求的整个生命周期. 通过 注解 的方式，相较于传统的 Controller，代码更简洁，用户可以更关注业务逻辑。

//

(1)、创建控制器

主要通过 @Controller 注解实现。代码可以放置任意位置，不过为了统一标准，建议放在 app/Http/Controller 下。

//

(2)、注解

///1/ Controller

Http 控制器类注解tag @Controller。

注解类：Swoft\Http\Server\Annotation\Mapping\Controller。

作用范围：CLASS。

拥有属性：prefix 指定路由前缀。

提示：通常仅有 @Controller 是没有什么效果的，它需要配合接下来的 @RequestMapping 一起才能正确的工作。

//

(3)、使用

显式指定路由前缀: @Controller(prefix="/index") 或 @Controller("/index")。

隐式指定路由前缀: @Controller() 默认自动解析 controller class 的名称，并且使用小驼峰格式。

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

/\*\*

 \* @Controller()

 \*/

class IndexController

{

    /\*\*

     \* @RequestMapping(route="index")

     \*/

    public function index(){}

}

//

// OR

//

/\*\*

 \* @Controller(prefix="/v1/index")

 \*/

class IndexController

{

    /\*\*

     \* @RequestMapping(route="index")

     \*/

    public function index(){}

}

?>

//// 启动服务器

隐式指定路由前缀访问URL：http://127.0.0.1:18306/index/index

显示指定路由前缀访问URL：http://127.0.0.1:18306/v1/index/index

//

(4)、说明

一个完整的路由规则是通过 @Controller + @RequestMapping 注解实现，通常前者定义前缀，后者定义后缀。

//

//// 如下，访问路由是 /v1/users/list (/v1/users + list)

/\*\*

 \* @Controller(prefix="/v1/users")

 \*/

class UsersController

{

    /\*\*

     \* @RequestMapping(route="list")

     \*/

    public function list(){}

}

访问URL：http://127.0.0.1:18306/v1/users/list

//

//// 当 @Controller 为空时

/\*\*

 \* @Controller()

 \*/

class UsersController

{

    /\*\*

     \* @RequestMapping(route="list")

     \*/

    public function list(){}

}

访问URL：http://127.0.0.1:18306/users/list

@Controller 和 @RequestMapping 的搭配使用。关于 @RequestMapping 注解详细参考：路由( http://swoft.org/docs/2.x/zh-CN/http-server/route.html )。

//

(5)、注意

在Swoft里不要按照传统的fpm框架继承父类控制器的成员属性在其他控制器使用，这种做法是错误的。

/\*\*

 \* @Controller()

 \*/

class BaseController

{

    protected $num;

}

/\*\*

 \* @Controller(prefix="/v1/index")

 \*/

class IndexController extends BaseController

{

    /\*\*

     \* @RequestMapping(route="index")

     \*/

    public function index()

    {

        $this->num++;

        echo $this->num."\n";

    }

}

--

4、路由绑定

( http://swoft.org/docs/2.x/zh-CN/http-server/route.html )

(0)、路由简述

Swoft与传统的PHP框架不一样，并没有采用配置文件的方式来配置路由，而采用了注解。在Swoft里我们可以使用 @RequestMapping 注解快速的添加路由。

//

(1)、路由器配置

///0/ 默认配置

// at file: vendor/swoft/http-server/src/AutoLoader.php

'httpRouter'      => [

    'name'            => 'swoft-http-router',

    // config

    'ignoreLastSlash' => true,

    'tmpCacheNumber'  => 500,

    // 'handleMethodNotAllowed' => false

],

你需要自定义路由配置，直接在 app/bean.php 添加 httpRouter 项配置即可。

'httpRouter'  => [

    'handleMethodNotAllowed' => true

]

//

///1/ 路由配置说明

ignoreLastSlash bool 默认：true 是否忽略URI path最后的 /，如果设置为 false 不忽略，/home 与 /home/ 将是两个不同的路由。

tmpCacheNumber int 默认：500 动态路由缓存数量，动态参数路由匹配后会缓存下来，下次相同的路由将会更快的匹配命中。

handleMethodNotAllowed bool 默认：false 是否处理 MethodNotAllowed，为了加快匹配速度，默认method不匹配也是直接抛出 Route not found 错误，如有特殊需要可以开启此选项，开启后将会抛出 Method Not Allowed 错误。

--

(2)、路由注解

///1/ RequestMapping 注解

// 可设置属性

route 路由规则path。

method 请求方式（GET、POST、PUT、PATCH、DELETE、OPTIONS、HEAD）。

params 可以通过它为path变量添加正则匹配限制。

注意：每个方法上尽量只写一个 @RequestMapping 注解，以免出现紊乱。

//

(3)、使用示例

通常情况，一个完整的路由path等于 Controller 的 prefix + RequestMapping 的 route。显示指定路由后缀：@RequestMapping("index") 或 @RequestMapping(route="index")；隐式指定路由后缀: 使用 @RequestMapping() 默认解析方法名为后缀。

特殊的，当你的 RequestMapping.route 上的路由以 / 开头时，那完整的路由就是它，即不会再将 prefix 添加到它的前面.

//

///1/ 简单使用

使用方法在控制器方法中加入 RequestMapping 注解。

/\*\*

 \* @Controller()

 \*/

class UserController

{

    /\*\*

     \* @RequestMapping()

     \*/

    public function index()

    {}

    /\*\*

     \* @RequestMapping("index")

     \*/

    public function index()

    {}

    /\*\*

     \* @RequestMapping(route="index")

     \*/

    public function index()

    {}

}

上面的效果一样，为index方法绑定的路由为 /user/index，允许的请求方法为默认的 GET 和 POST。

//

///2/ 绑定路由path参数

指定路由参数: @RequestMapping(route="index/{name}")，Action 方法中可以直接使用 $name 作为方法参数。

当路由参数被 [] 包起来则URL path传递参数是可选的，可有可无。

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

/\*\*

 \* @Controller()

 \*/

class UserController

{

    /\*\*

     \* @RequestMapping(route="/index/{name}")

     \*/

    public function index(string $name)

    {}

    /\*\*

     \* @RequestMapping(route="/index/{name}")

     \*/

    public function index(string $name)

    {}

}

?>

//

///3/ 设置method参数

如果想要设置允许请求控制器的HTTP请求方式。 可以使用方法在控制器中的 @RequestMapping 注解配置method参数，可以是 GET、POST、PUT、PATCH、DELETE、OPTIONS、HEAD 中的一个或多个。

/// 限定HTTP方法: @RequestMapping(method={RequestMethod::GET}) 指定路由支持的HTTP方法，默认是支持 GET 和 POST。比如 method={RequestMethod::POST,RequestMethod::PUT} 设置路由支持 POST 和 PUT。

<?php declare(strict\_types=1);

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

use Swoft\Http\Server\Annotation\Mapping\RequestMethod;

/\*\*

 \* @Controller()

 \*/

class UserController

{

    /\*\*

     \* @RequestMapping(method=RequestMethod::GET)

     \*/

    public function index()

    {}

    /\*\*

     \* @RequestMapping(method={RequestMethod::GET,RequestMethod::POST})

     \*/

    public function index()

    {}

    /\*\*

     \* @RequestMapping("index", method=RequestMethod::PUT)

     \*/

    public function index()

    {}

}

?>

//

(4)、使用注意

//// 请切记要引入相关的注解类

Swoft\Http\Server\Annotation\Mapping\RequestMapping

Swoft\Http\Server\Annotation\Mapping\RequestMethod

//

(5)、获取匹配结果

你可以在中间件或者action拿到路由匹配的结果信息。

[$status, $path, $route] = $request->getAttribute(Request::ROUTER\_ATTRIBUTE);

--

5、请求对象

( http://swoft.org/docs/2.x/zh-CN/http-server/request.html )

Swoft 的请求与响应实现于 PSR 7 规范。请求与响应对象存在于每次 HTTP 请求。

请求对象 Request 为 Swoft\Http\Message\Request。

响应对象 Response 为 Swoft\Http\Message\Response。

注意！根据PSR-7对象的不可变性(immutable)，所有的 with\* 方法都是克隆对象然后返回，必须接收新对象来做进一步处理，或使用链式调用。

//

(1)、基本方法

///1/ 公共方法

// PSR-7 接口为请求和响应对象提供了这些公共方法。

withProtocolVersion($version)

withHeader($name, $value)

withAddedHeader($name, $value)

withoutHeader($name)

withBody(StreamInterface $body)

//

///2/ 请求对象方法

// PSR-7 接口为请求对象提供了这些方法。

withMethod(string $method)

withUri(UriInterface $uri, $preserveHost = false)

withCookieParams(array $cookies)

withQueryParams(array $query)

withUploadedFiles(array $uploadedFiles)

withParsedBody($data)

withAttribute($name, $value)

withoutAttribute($name)

更多请参考 PSR-7 和 查看 swoft/http-message 中具体的实现类( https://github.com/swoft-cloud/swoft-http-message )。

//

(2)、如何获取

通过控制器方法参数注入 (Request $request)。

通过请求上下文获取 Swoft\Context\Context::mustGet()->getRequest()。

//

(3)、请求动作方法

$request = context()->getRequest();

$method = $request->getMethod();

//

(4)、请求的URI

每个 HTTP 请求都有一个URI标识所请求的应用程序资源。HTTP 请求 URI 有几个部分。

Scheme (e.g. http or https)

Host (e.g. example.com)

Port (e.g. 80 or 443)

Path (e.g. /users/1)

Query string (e.g. sort=created&dir=asc)

你可以通过请求对象的 getUri() 方法获取 PSR-7 URI对象( http://www.php-fig.org/psr/psr-7/#3-5-psr-http-message-uriinterface )。

//

$uri = $request->getUri();

PSR-7 请求对象的 URI 本身就是一个对象,它提供了下列方法检查 HTTP 请求的 URL 部分。

getScheme()

getAuthority()

getUserInfo()

getHost()

getPort()

getPath()

getQuery() (e.g. a=1&b=2)

getFragment()

//

(5)、请求 Headers

///1/ 全部的 Headers

$headers = $request->getHeaders();

foreach ($headers as $name => $values) {

    echo $name . ": " . implode(", ", $values).PHP\_EOL;

}

//

///2/ 指定的 Header

/// 返回值是array

$headerValueArray = $request->getHeader('host');

print\_r($headerValueArray);

//

/// 返回值是字符串

$host = $request->getHeaderLine("host");

print\_r($host);

//

(6)、请求数据获取

///1/ GET 数据

$data = $request->query();

$some = $request->query('key', 'default value')

$data = $request->get();

$some = $request->get('key','default value');

//

///2/ POST 数据

$data = $request->post();

$some = $request->post('key', 'default value')

提示： 无需关心请求的数据格式，json xml 请求都会自动解析为php的数组数据。都可以通过 $request->post() 获取。

//

///3/ GET & POST 数据

$data = $request->input();

$some = $request->input('key', 'default value')

//

///4/ RAW 数据

$data = $request->raw();

//

///5/ SERVER 数据

$data = $request->getServerParams();

$some = $request->server('key', 'default value')

//

///6/ 获取上传文件

$file = $request->getUploadedFiles();

获取的结果是一维数组或者二位数组，数据结构如下。 若表单中上传的是单文件则返回的是一个一维数组，数组内容是 Swoft\Http\Message\Upload\UploadedFile 文件对象，例如文件字段名为 file 则数据结构为如下所示。

array(1) {

  ["file"]=>

  object(Swoft\Http\Message\Upload\UploadedFile)#6510 (7) {

    ["size":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    int(1319)

    ["errorCode":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    int(0)

    ["file":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    string(25) "/tmp/swoole.upfile.f7p2EL"

    ["clientFilename":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    string(6) "at.png"

    ["clientMediaType":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    string(9) "image/png"

    ["moved":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    NULL

    ["path":"Swoft\Http\Message\Upload\UploadedFile":private]=>

    NULL

  }

}

若表单中是一个字段数组上传多个文件如 file[] 则返回的是一个二维数组，数组内容依然是 Swoft\Http\Message\Upload\UploadedFile 文件对象，数据结构如下所示。

array(1) {

  ["file"]=>

  array(2) {

    [0]=>

    object(Swoft\Http\Message\Upload\UploadedFile)#6516 (7) {

      ["size":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      int(1319)

      ["errorCode":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      int(0)

      ["file":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      string(25) "/tmp/swoole.upfile.TVKdOS"

      ["clientFilename":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      string(6) "at.png"

      ["clientMediaType":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      string(9) "image/png"

      ["moved":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      NULL

      ["path":"Swoft\Http\Message\Upload\UploadedFile":private]=>

      NULL

    }

    ...

  }

}

//

(7)、文件操作方法

moveTo() 将上传的文件移动到新位置。

getSize() 获取文件大小，单位 byte。

getError() 获取上传文件相关的错误信息，若无错将必须返回UPLOAD\_ERR\_OK 常量，若有错误将返回UPLOAD\_ERR\_XXX 相关常量。

getClientFilename() 获取文件上传时客户端本地的文件名，不要相信此方法返回的值。客户端可能会发送恶意虚假文件名，意图破坏或破解您的应用程序。

getClientMediaType() 获取客户端中文件的 MediaType 类型，不要相信此方法返回的值。客户端可能会发送恶意虚假文件名，意图破坏或破解您的应用程序。

//

(8)、一些辅助方法

///1/ XHR

if ($request->isAjax()) {

    // Do something

}

if ($request->isXmlHttpRequest()) {

    // Do something

}

//

///2/ GET

if ($request->isGet()) {

    // Do something

}

//

///3/ POST

if ($request->isPost()) {

    // Do something

}

//

///4/ PUT

if ($request->isPut()) {

    // Do something

}

//

///5/ DELETE

if ($request->isDelete()) {

    // Do something

}

//

///6/ PATCH

if ($request->isPatch()) {

    // Do something

}

//

///7/ Content-Type

$contentType = $request->getContentType();

--

6、响应对象

( http://swoft.org/docs/2.x/zh-CN/http-server/response.html )

Swoft 的请求与响应实现于 PSR 7 规范。请求与响应对象存在于每次 HTTP 请求。

请求对象 Request 为 Swoft\Http\Message\Request。

响应对象 Response 为 Swoft\Http\Message\Response。

注意！根据PSR-7对象的不可变性(immutable)，所有的 with\* 方法都是克隆对象然后返回，必须接收新对象来做进一步处理，或使用链式调用。

//

(1)、基本方法

///1/ 公共方法

/// PSR-7 接口为请求和响应对象提供了这些公共方法。

withProtocolVersion($version)

withHeader($name, $value)

withAddedHeader($name, $value)

withoutHeader($name)

withBody(StreamInterface $body)

//

///2/ 响应对象

/// PSR-7 接口为响应对象提供了这些方法。

withStatus($code, $reasonPhrase = '')

更多请参考 PSR-7 和 查看 swoft/http-message( https://github.com/swoft-cloud/swoft-http-message ) 中具体的实现类。

//

(2)、如何获取

通过控制器方法参数注入 (Response $response)。

通过请求上下文获取 context()->getResponse()。

通过请求上下文获取 Swoft\Context\Context::mustGet()->getResponse() (已废弃)。

//

(3)、设置状态码

$response = \context()->getResponse();

return $response->withStatus(404);

//

(4)、输出字符串内容

return $response->withContent("Hello Swoft2.0");

//

(5)、输出数组

$data = ['name'=>'Swoft2.0'];

$response->withData($data);

//

(6)、输出头信息

return $response->withHeader("name","Swoft2.0");

//

(7)、重定向

// 302

return $response->redirect("http://www.swoft.org",302);

// 404 page

return $response->redirect('/404');

//

(8)、文件下载

return $response->file(\alias('@runtime/1.zip'), "application/octet-stream");

//

(9)、设置Cookies

$response = $response->withCookie(’name', 'value');

$response = $response->withCookie(’name', [

    'value'    => 'value3',

    'httpOnly' => true

]);

// 设置多个

$cookies = [

    'key1' => 'value1',

    'key2' => [

        'value' => 'value2',

    ],

    'key3' => [

        'value'    => 'value3',

        'httpOnly' => true

    ],

];

$response = $response->withCookies($cookies);

--

7、中间件

( http://swoft.org/docs/2.x/zh-CN/http-server/middleware.html )

(1)、声明一个中间件

中间件是用于控制 请求到达 和 响应请求 的整个流程的，通常用于对请求进行过滤验证处理，当你需要对请求或响应作出对应的修改或处理，或想调整请求处理的流程时均可以使用中间件来实现。

//

(2)、定义中间件

只需要实现了 Swoft\Http\Server\Contract\MiddlewareInterface 接口均为一个合法的中间件，其中 process() 方法为该中间件逻辑处理方法。

<?php

namespace App\Http\Middleware;

use Psr\Http\Message\ResponseInterface;

use Psr\Http\Message\ServerRequestInterface;

use Psr\Http\Server\RequestHandlerInterface;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Http\Server\Contract\MiddlewareInterface;

/\*\*

 \* @Bean()

 \*/

class ControllerMiddleware implements MiddlewareInterface

{

    /\*\*

     \* Process an incoming server request.

     \*

     \* @param ServerRequestInterface $request

     \* @param RequestHandlerInterface $handler

     \*

     \* @return ResponseInterface

     \* @inheritdoc

     \*/

    public function process(ServerRequestInterface $request, RequestHandlerInterface $handler): ResponseInterface

    {

        $response = $handler->handle($request);

        return $response;

    }

}

?>

//

(3)、配置全局中间件

当你的自定义中间件需要全局请求应用，则可以考虑将此中间件作为全局中间件去使用，只需在 Bean 配置文件内配置 httpDispatcher 的 middlewares 属性，在数组中加入你的自定义中间件的命名空间地址，相关配置通常在 app/bean.php 内。

return [

    ...

    'httpDispatcher'=>[

        'middlewares'=>[

            AuthMiddleware::class,

            ApiMiddleware::class

        ]

    ]

    ...

]

//

(4)、通过注解使用

通过 @Middleware 和 @Middlewares, 可以很方便的配置中间件到当前的 Controller 和 Action 内。

当将此注解应用于 Controller 上，则作用域为整个 Controller。

将此注解应用于 Action 上，则作用域仅为当前的 Action。

@Middleware 用于配置单个中间件。

@Middlewares 显而易见的是用于配置一组 @Middleware，按照定义顺序依次执行。

<?php

namespace App\Http\Controller;

use App\Http\Middleware\ApiMiddleware;

use App\Http\Middleware\IndexMiddleware;

use App\Http\Middleware\ControllerMiddleware;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\Middleware;

use Swoft\Http\Server\Annotation\Mapping\Middlewares;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

/\*\*

 \* @Controller()

 \* @Middlewares({

 \*      @Middleware(ApiMiddleware::class),

 \*      @Middleware(ControllerMiddleware::class)

 \* })

 \*/

class MiddlewareController

{

    /\*\*

     \* @RequestMapping()

     \* @Middleware(IndexMiddleware::class)

     \*/

    public function index(){

        return "MiddlewareController";

    }

}

?>

注意：记得要引入对应的中间件类。

//

(5)、应用

///1/ 提前拦截请求

注意： 拦截要在 $handler->handle($request) 之前。

<?php

namespace App\Http\Middleware;

use Psr\Http\Message\ResponseInterface;

use Psr\Http\Message\ServerRequestInterface;

use Psr\Http\Server\RequestHandlerInterface;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Context\Context;

use Swoft\Http\Server\Contract\MiddlewareInterface;

/\*\*

 \* @Bean()

 \*/

class SomeMiddleware implements MiddlewareInterface

{

    /\*\*

     \* Process an incoming server request.

     \* @param ServerRequestInterface $request

     \* @param RequestHandlerInterface $handler

     \* @return ResponseInterface

     \* @inheritdoc

     \*/

    public function process(ServerRequestInterface $request, RequestHandlerInterface $handler): ResponseInterface

    {

        $path = $request->getUri()->getPath();

        if ($path === '/favicon.ico') {

            $response = Context::mustGet()->getResponse();

            return $response->withStatus(404);

        }

        return $handler->handle($request);

    }

}

?>

//

///2/ 跨域设置

<?php

namespace App\Http\Middleware;

use Psr\Http\Message\ResponseInterface;

use Psr\Http\Message\ServerRequestInterface;

use Psr\Http\Server\RequestHandlerInterface;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Http\Server\Contract\MiddlewareInterface;

/\*\*

 \* @Bean()

 \*/

class CorsMiddleware implements MiddlewareInterface

{

    /\*\*

     \* Process an incoming server request.

     \* @param ServerRequestInterface $request

     \* @param RequestHandlerInterface $handler

     \* @return ResponseInterface

     \* @inheritdoc

     \*/

    public function process(ServerRequestInterface $request, RequestHandlerInterface $handler): ResponseInterface

    {

        if ('OPTIONS' === $request->getMethod()) {

            $response = Context::mustGet()->getResponse();

            return $this->configResponse($response);

        }

        $response = $handler->handle($request);

        return $this->configResponse($response);

    }

    private function configResponse(ResponseInterface $response)

    {

        return $response

            ->withHeader('Access-Control-Allow-Origin', 'http://mysite')

            ->withHeader('Access-Control-Allow-Headers', 'X-Requested-With, Content-Type, Accept, Origin, Authorization')

            ->withHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');

    }

}

?>

//

///3/ 用户JWT登陆验证

<?php

namespace App\Http\Middleware;

use Psr\Http\Message\ResponseInterface;

use Psr\Http\Message\ServerRequestInterface;

use Psr\Http\Server\RequestHandlerInterface;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Context\Context;

use Swoft\Http\Server\Contract\MiddlewareInterface;

/\*\*

 \* @Bean()

 \*/

class AuthMiddleware implements MiddlewareInterface

{

    /\*\*

     \* Process an incoming server request.

     \* @param ServerRequestInterface $request

     \* @param RequestHandlerInterface $handler

     \* @return ResponseInterface

     \* @inheritdoc

     \*/

    public function process(ServerRequestInterface $request, RequestHandlerInterface $handler): ResponseInterface

    {

        // before request handle

        // 判断token

        $token = $request->getHeaderLine("token");

        $type = \config('jwt.type');

        $public = \config('jwt.publicKey');

        try {

            $auth = JWT::decode($token, $public, ['type' => $type]);

            $request->user = $auth->user;

        } catch (\Exception $e) {

            $json = ['code'=>0,'msg'=>'授权失败']

            $response = Context::mustGet()->getResponse();

            return $response->withData($json);

        }

        $response = $handler->handle($request);

        return $response;

        // after request handle

    }

}

?>

--

8、异常处理

( http://swoft.org/docs/2.x/zh-CN/http-server/exception.html )

通常我们把异常类放置 app/Exception ，异常类处理器放置 app/Exception/Handler。异常分为两部分。自定义的 Exception 异常类，异常处理类 ExceptionHandler。

//

(1)、定义异常类

在不同应用场景下，定义不同的异常类，如需要一个控制器抛异常的类app/Exception/ControllerException.php。

namespace App\Exception;

class ApiException extends \Exception

{

}

//

(2)、定义异常处理类

<?php

namespace App\Exception\Handler;

use App\Exception\ApiException;

use Swoft\Error\Annotation\Mapping\ExceptionHandler;

use Swoft\Http\Message\Response;

use Swoft\Http\Server\Exception\Handler\AbstractHttpErrorHandler;

/\*\*

 \* @ExceptionHandler(ApiException::class)

 \*/

class ApiExceptionHandler extends AbstractHttpErrorHandler

{

    /\*\*

     \* @param \Throwable $e

     \* @param Response $response

     \* @return Response

     \* @throws \ReflectionException

     \* @throws \Swoft\Bean\Exception\ContainerException

     \*/

    public function handle(\Throwable $e, Response $response): Response

    {

        $data = ['code'=>-1,'msg'=>$e->getMessage()];

        return $response->withData($data);

    }

}

?>

//

(3)、注解

///1/ ExceptionHandler

异常处理程序，指定这个处理器要处理当异常，当程序抛出 ExceptionHandler 注解里有的异常将会自动执行 handle 方法。

//// 指定异常：参数可以是字符串也可以是数组。

/// 处理一个异常

@ExceptionHandler(ApiException::class)

/// 处理多个异常

@ExceptionHandler({ApiException::class,ServiceException::class})

--

9、HTTP 客户端

( http://swoft.org/docs/2.x/zh-CN/http-server/http.html )

Http Client 官方建议使用 saber 和 Guzzle ，不再重复造轮子。

//

(1)、Saber

Swoole 官方封装的 Http client 库，已在多个大型项目中使用。

//

///1/ 安装

composer require swlib/saber

//

///2/ 使用

SaberGM::get('http://httpbin.org/get');

SaberGM::delete('http://httpbin.org/delete');

SaberGM::post('http://httpbin.org/post', ['foo' => 'bar']);

SaberGM::put('http://httpbin.org/put', ['foo' => 'bar']);

SaberGM::patch('http://httpbin.org/patch', ['foo' => 'bar']);

//

(2)、Guzzle

Guzzle 老牌 HTTP 封装库，很多依赖都是它封装的。swoole 版本必须 4.4 即以上。

//

///1/ 安装 (2019-10-04 ^6.3)

composer require guzzlehttp/guzzle

//

///2/ 使用

$client = new \GuzzleHttp\Client();

$response = $client->request('GET', 'https://api.github.com/repos/guzzle/guzzle');

echo $response->getStatusCode(); # 200

echo $response->getHeaderLine('content-type'); # 'application/json; charset=utf8'

echo $response->getBody(); # '{"id": 1420053, "name": "guzzle", ...}'

--

--

三、Websocket

--

0、WebSocket 服务

( http://swoft.org/docs/2.x/zh-CN/websocket-server/index.html )

WebSocket 服务基于现有 swoole ws server上的进一步封装实现。即开启websocket服务的同时可以处理http请求。

//

(1)、安装

composer require swoft/websocket-server

//

(2)、功能特色

快速的搭建使用 websocket server；

支持完全的自定义流程处理，如果你不想使用框架自带的处理；

支持消息阶段的数据解析和路由调度；

通用的消息发送方法封装(send, sendToSome, sendToAll, broadcast 等)；

//

(3)、连接处理流程

握手请求 -------> 接收到WebSocket请求(根据path找到处理模块 eg EchoModule)

                         |

                         |使用模块类中标记的握手方法验证请求

                         |

                         v

                   握手成功，接受连接

                         |

                         | 创建连接上下文Connection，存储到Session管理器

                         | (含有fd, request等信息)

                         |

                         v

          消息请求 --> 接收消息

                         |

                         |创建消息上下文Context，存储到Context管理器

                         |同时通过CoID会与Session的绑定关系

                         |

                         V

                    解析消息数据

                         |

                         |得到消息指令和消息body

                         |(根据消息指令找到处理控制器 eg ChatController)

                         |

                         V

                    调度消息处理

                         |

                         |调用对应的message控制器方法处理

                         |

                         V

                    打包返回数据

                         |

                         |销毁此次消息请求的上下文Context

                         |同时删除与Session的绑定关系

                         |

                         v

          得到响应 <--- 返回结果

                         |

          消息请求 -->    |

              .          |

              .          |(重复上述消息处理流程)

              .          |

          <-- 得到响应    |

                         |

                         v

断开连接 -——-——-->  收到关闭连接事件

                         |

                         | 销毁连接上下文，从Session管理器删除此连接

                         |

                         v

                      关闭连接

//

(4)、Git仓库

github https://github.com/swoft-cloud/swoft-websocket-server

//

(5)、参与贡献

欢迎参与贡献，您可以fork 我们的开发仓库 swoft/component( https://github.com/swoft-cloud/swoft-component )；修改代码然后发起 PR；关于发起PR的注意事项( https://github.com/swoft-cloud/swoft/issues/829 )。

--

1、配置服务 (websocket 配置)

( http://swoft.org/docs/2.x/zh-CN/websocket-server/config.html )

websocket 的 host, port 等配置是都是完全可以自定义的。 配置需要编辑 app/bean.php 文件，下面列举了一些简单的配置，你也可以自由组合同时提供多种服务。websocket server 的默认端口是 18308。

//

(1)、可配置项

///1/ 可配置项用于 ws server bean 配置，除了 class 其他都是 ws server 的属性。

class 指定 websocket server 的处理类。

port 指定 websocket server 的端口。

listener 指定其他一同启动的服务，添加端口服务监听，可以多个，rpc 启动 RPC 服务。

on 配置监听的事件，注册事件、设置对应事件的处理监听，事件触发组件调用，在任务里面使用。

setting 这里是参考 Swoole Server配置选项(https://wiki.swoole.com/wiki/page/274.html)。

pidFile 设置进程 pid文件 位置，默认值 @runtime/swoft.pid。

mode 运行的模式，参考 Swoole Server 构造函数( https://wiki.swoole.com/wiki/page/14.html ) 第三个参数。

type 指定Socket的类型，支持TCP、UDP、TCP6、UDP6、UnixSocket Stream/Dgram 等 Swoole Server 构造函数( https://wiki.swoole.com/wiki/page/14.html ) 第四个参数。

//

(2)、基础配置

// ...

'wsServer'   => [

    'class'   => WebSocketServer::class,

    'port' => 18307,

    'debug' => env('SWOFT\_DEBUG', 0),

    /\* @see WebSocketServer::$setting \*/

    'setting' => [

        'log\_file' => alias('@runtime/swoole.log'),

    ],

],

//

(3)、启用http请求处理

默认的是没有启用http server功能的。如果你想开启ws时，同时处理http请求。

// ...

'wsServer'   => [

    'class'   => WebSocketServer::class,

    'on'      => [

        // 加上如下一行，开启处理http请求

        SwooleEvent::REQUEST => bean(RequestListener::class),

    ],

    'debug' => env('SWOFT\_DEBUG', 0),

    /\* @see WebSocketServer::$setting \*/

    'setting' => [

        'log\_file' => alias('@runtime/swoole.log'),

    ],

],

ok, 现在 IP:PORT 上可以同时处理 http 和 ws 请求了。

//

(4)、启用wss支持

跟在http server启用https类似，在swoft里启用 wss 也非常简单，添加如下的配置即可。

'httpServer' => [

    'type' => SWOOLE\_SOCK\_TCP | SWOOLE\_SSL,

    /\* @see WebSocketServer::$setting \*/

    'setting'  => [

        'ssl\_cert\_file' => '/my/certs/2288803\_www.domain.com.pem',

        'ssl\_key\_file'  => '/my/certs/2288803\_www.domain.com.key',

    ]

]

注意： 你必须安装 OpenSSL 库，并且确保安装swoole时是启用了 ssl 选项的。同时，需要设置 'type' => SWOOLE\_SOCK\_TCP | SWOOLE\_SSL。

//

(5)、添加RPC服务

如果你想开启ws时，同时启动RPC Server服务。

// ...

'wsServer'   => [

    'listener' => [

        'rpc' => \bean('rpcServer') // 引入 rpcServer

    ],

],

'rpcServer'  => [

    'class' => ServiceServer::class,

    'port' => 18308,

],

//

(6)、启用全部功能；

websocket server；

http server；

task process；

rpc server；

//

// ...

'wsServer'   => [

    'class'   => WebSocketServer::class,

    'port' => 18307,

    'on'      => [

        // 开启处理http请求支持

        SwooleEvent::REQUEST => bean(RequestListener::class),

        // 启用任务必须添加 task, finish 事件处理

        SwooleEvent::TASK   => bean(TaskListener::class),

        SwooleEvent::FINISH => bean(FinishListener::class)

    ],

    'listener' => [

        // 引入 rpcServer

        'rpc' => \bean('rpcServer')

    ],

    'debug' => env('SWOFT\_DEBUG', 0),

    /\* @see WebSocketServer::$setting \*/

    'setting' => [

        'log\_file' => alias('@runtime/swoole.log'),

        // 任务需要配置 task worker

        'task\_worker\_num'       => 2,

        'task\_enable\_coroutine' => true

    ],

],

'rpcServer'  => [

    'class' => ServiceServer::class,

    'port' => 18308,

],

ok, 现在通过 php bin/swoft ws:start 启动的服务器，就支持上面的全部功能了。

--

2、管理服务

( http://swoft.org/docs/2.x/zh-CN/websocket-server/manage.html )

(0)、启动与管理

在项目目录下执行如下命令可以看到websocket server的管理命令. 跟http server的管理命令一致.

$ php bin/swoft ws

Description:

  There some commands for manage the webSocket server

Usage:

  ws:{command} [arguments] [options]

Commands:

  start    Start the webSocket server

  stop     Stop the running server

  restart  Restart the running server

Options:

  -h, --help  Show help of the command group or specified command action

//

(1)、启动

///1/ 前台运行

$ php bin/swoft ws:start

//

///2/ 后台运行

$ php bin/swoft ws:start -d

//

(2)、使用

如果你注册了ws的路由处理模块，现在就可以通过浏览器等ws客户端连接上server了。

--

3、定义模块

( http://swoft.org/docs/2.x/zh-CN/websocket-server/module.html )

在根据上两章安装配置好之后，就可以在 app/WebSocket 下创建需要的 websocket 模块来处理相关逻辑。在每个模块里允许用户处理的几个事件有 handshake open message close。

//

(1)、注解

///1/ WsModule

websocket 模块类注解tag @WsModule。

注解类：Swoft\WebSocket\Server\Annotation\Mapping\WsModule。

作用范围：CLASS。

/// 拥有属性

path string 标明了允许ws连接的URI path。

controllers array 绑定到此模块的消息控制器类。

messageParser string 绑定到此模块的消息数据解析器。

defaultOpcode integer 此模块默认的消息数据 opcode。

//

/// 示例

/\*\*

 \* @WsModule("/echo")

 \*/

上面的注解标明了允许ws连接的URI path，即客户端请求的ws连接类似：ws://IP:PORT/echo。

//

///2/ OnHandshake

方法注解 @OnHandshake 标记处理握手的方法。

注解类： Swoft\WebSocket\Server\Annotation\Mapping\OnHandshake。

作用范围： METHOD。

提示：这方法是可选的，如果没有特殊的需求，可以忽略它，框架会帮你握手并响应握手成功。

//

/// 必须返回含有两个元素的array

bool 第一个元素的值来决定是否进行握手。

第二个元素是response对象 - 可以在response设置一些自定义header，body等信息。

//

///3/ OnOpen

在握手成功后，就会触发 open 事件. 方法注解 @OnOpen 标记对应方法。(此时开始你就可以给客户端发消息了)

注解类：Swoft\WebSocket\Server\Annotation\Mapping\OnOpen。

作用范围：METHOD。

此方法也是可选的，可以没有。

//

///4/ OnMessage

通过的方法注解 @OnMessage 标记一个消息处理方法。在此阶段你可以接收到客户端的消息和发送消息给对方。

注解类：Swoft\WebSocket\Server\Annotation\Mapping\OnMessage。

作用范围：METHOD。

当你没有绑定消息控制器时，表明你想自己处理消息阶段的逻辑， 此方法是必须存在的。

当你有绑定消息控制器时，框架会自动解析消息并路由到指定的消息处理方法。

//

///5/ OnClose

通过的方法注解 @OnClose 标记一个关闭连接时的处理方法。当客户的关闭连接或者server在其他地方主动关闭连接时，就会触发此事件。你可以在这里做一些连接关闭后的工作, 比如：记录日志，解绑用户等。

注解类：Swoft\WebSocket\Server\Annotation\Mapping\OnClose。

作用范围：METHOD。

此方法也是可选的，可以没有。

注意：触发此事件时连接已被关闭，不能再给对方发消息。

//

(2)、代码示例

///1/ 这里面方法上的 server 对象都是 Swoole\WebSocket\Server 的实例

<?php

namespace App\WebSocket;

use Swoft\Http\Message\Request;

use Swoft\Http\Message\Response;

use Swoft\WebSocket\Server\Annotation\Mapping\OnClose;

use Swoft\WebSocket\Server\Annotation\Mapping\OnHandshake;

use Swoft\WebSocket\Server\Annotation\Mapping\OnOpen;

use Swoft\WebSocket\Server\Annotation\Mapping\WsModule;

use Swoole\WebSocket\Frame;

use Swoole\WebSocket\Server;

/\*\*

 \* Class EchoModule

 \*

 \* @WsModule("/echo")

 \*/

class EchoModule

{

    /\*\*

     \* 在这里你可以验证握手的请求信息

     \* @OnHandshake()

     \* @param Request $request

     \* @param Response $response

     \* @return array [bool, $response]

     \*/

    public function checkHandshake(Request $request, Response $response): array

    {

        return [true, $response];

    }

    /\*\*

     \* On connection has open

     \*

     \* @OnOpen()

     \* @param Request $request

     \* @param int     $fd

     \*/

    public function onOpen(Request $request, int $fd): void

    {

        server()->push($fd, 'hello, welcome! :)');

    }

    /\*\*

     \* @OnMessage()

     \* @param Server $server

     \* @param Frame $frame

     \*/

    public function onMessage(Server $server, Frame $frame)

    {

        $server->push($frame->fd, 'I have received message: ' . $frame->data);

    }

    /\*\*

     \* On connection closed

     \* - you can do something. eg. record log

     \*

     \* @OnClose()

     \* @param Server $server

     \* @param int    $fd

     \*/

    public function onClose(Server $server, int $fd): void

    {

        // you can do something. eg. record log, unbind user...

    }

}

?>

//

(3)、客户端测试 (其实可以使用http页面WebSocket测试)

如果你安装并启用了 devtool, 那么你可以打开页面 IP:PORT/\_\_devtool/ws/test 来进行ws测试。

填上你的ws server地址(注意不要忘了URI path)。然后就可以连接上ws server 并收发消息了。如果你在前台运行的server 你也能在运行 server的console 上看到ws连接与消息log。

提示：当然也可在网上找一个 ws test网页来进行测试。

//

///1/ 使用http页面WebSocket测试

<div id="div-ws">

    <a href="javascript:webSocketTest()">运行WebSocket客户端("ws://127.0.0.1:18308/echo")</a>

</div>

<script type="text/javascript">

    //// websocket客户端

    function webSocketTest()

    {

        //// WebSocket服务检测

        if ("WebSocket" in window)

        {

            log("您的浏览器支持 WebSocket!");

            ///0/ 打开websocket客户端

            var wsUrl = "ws://127.0.0.1:18308/echo";

            var ws = new WebSocket(wsUrl);

            //

            ///1/ 连接监听

            ws.onopen = function()

            {

                // WebSocket 已连接上，使用 send() 方法发送数据

                ws.send("客户端发送数据;");

                log("客户端数据发送中...;");

            };

            //

            ///2/ 监听信息

            ws.onmessage = function (evt)

            {

                // 接收信息

                var receivedMsg = evt.data;

                log("客户端接收服务器数据...["+receivedMsg+"];");

            };

            //

            ///3/ 监听关闭

            ws.onclose = function()

            {

                // 关闭 websocket

                log("连接已关闭...;");

            };

        }

        else {

            // 浏览器不支持 WebSocket

            log("您的浏览器不支持 WebSocket!");

        }

    }

    //// 日志输出

    function log(msg) {

        console.log(msg);

    }

</script>

--

4、消息控制器 (消息处理)

( http://swoft.org/docs/2.x/zh-CN/websocket-server/message-route.html )

// swoft 提供了灵活的 websocket 使用，支持自定义和由框架托管处理消息两种方式。

如果你在ws模块类没有添加 @OnMessage 处理方法，框架将会自动托管这个阶段，解析消息并根据路由分发到不同的方法执行。

如果你在ws模块类里面绑定了 @OnMessage 处理方法，swoft就认为你想自己处理这个阶段，框架就不会处理了。

本篇文档的使用是建立在由框架托管消息路由的基础上。

//

(1)、注解

///1/ WsController

websocket 消息控制器注解tag @WsController。

注解类：Swoft\WebSocket\Server\Annotation\Mapping\WsController。

作用范围：CLASS。

拥有属性：prefix string 消息路由前缀。

//

///2/ MessageMapping

方法注解 @MessageMapping 标记具体的消息处理方法，类似于http控制器里的action。

注解类：Swoft\WebSocket\Server\Annotation\Mapping\MessageMapping。

作用范围：METHOD。

拥有属性：command string 消息命令名称。

完整的消息路由path是 上面的 prefix 和 command 由点拼接而成 PREFIX.COMMAND。

//

(2)、消息解析

不同的使用者或者使用场景，用于ws通信的数据格式可能是不一样的。因此，在编写ws模块时，需要你绑定消息解析器。

//

///1/ 内置解析器

Swoft\WebSocket\Server\MessageParser\RawTextParser 简单的字符串。

Swoft\WebSocket\Server\MessageParser\TokenTextParser 简单的token字符串协议(方便测试使用的)。

Swoft\WebSocket\Server\MessageParser\JsonParser 简单的 json 数据协议。

//

///2/ JOSN 协议通信数据结构

{

    "cmd": "message route path. eg: home.index", // type: string

    "data": "message data", // type: mixed

    "ext": "message extea data", // optional, type: mixed

}

//

(3)、示例

///1/ 定义ws模块

注意 要绑定消息处理控制器，通常也需要绑定你的消息解析器，可以使用内置的几个简单的解析器，也可以根据需要自定义。

<?php declare(strict\_types=1);

namespace App\WebSocket;

use App\WebSocket\Chat\HomeController;

use Swoft\Http\Message\Request;

use Swoft\WebSocket\Server\Annotation\Mapping\OnOpen;

use Swoft\WebSocket\Server\Annotation\Mapping\WsModule;

use Swoft\WebSocket\Server\MessageParser\TokenTextParser;

use function server;

/\*\*

 \* Class ChatModule

 \*

 \* @WsModule(

 \*     "/chat",

 \*     messageParser=TokenTextParser::class,

 \*     controllers={HomeController::class}

 \* )

 \*/

class ChatModule

{

    /\*\*

     \* @OnOpen()

     \* @param Request $request

     \* @param int     $fd

     \*/

    public function onOpen(Request $request, int $fd): void

    {

        server()->push($request->getFd(), "Opened, welcome!(FD: $fd)");

    }

}

?>

定义的ws模块路径为 /chat。

绑定了的控制器有： HomeController::class。

绑定了一个内置的消息解析器。

注意：这里定义Ws模块时，绑定了一个框架自带的消息解析器，TokenTextParser::class 内置了一个decode 的方法用来解析数据。

// 默认为字符串解析，消息路由格式 `控制器.方法:数据`

public function decode(string $data): Message

{

    // use default message command

    $cmd = '';

    if (strpos($data, ':')) {

        [$cmd, $body] = explode(':', $data, 2);

        $cmd = trim($cmd);

    } else {

        $body = $data;

    }

    return Message::new($cmd, $body);

}

//

///2/ 消息控制器

注意 必须使用注解 @WsController 以及 @MessageMapping。

<?php declare(strict\_types=1);

namespace App\WebSocket\Chat;

use Swoft\Session\Session;

use Swoft\WebSocket\Server\Annotation\Mapping\MessageMapping;

use Swoft\WebSocket\Server\Annotation\Mapping\WsController;

/\*\*

 \* Class HomeController

 \*

 \* @WsController()

 \*/

class HomeController

{

    /\*\*

     \* Message command is: 'home.index'

     \*

     \* @return void

     \* @MessageMapping()

     \*/

    public function index(): void

    {

        Session::mustGet()->push('hi, this is home.index');

    }

    /\*\*

     \* Message command is: 'home.echo'

     \*

     \* @param string $data

     \* @MessageMapping()

     \*/

    public function echo(string $data): void

    {

        Session::mustGet()->push('(home.echo)Recv: ' . $data);

    }

    /\*\*

     \* Message command is: 'home.ar'

     \*

     \* @param string $data

     \* @MessageMapping("ar")

     \*

     \* @return string

     \*/

    public function autoReply(string $data): string

    {

        return '(home.ar)Recv: ' . $data;

    }

}

?>

注意，自 v2.0.6 版本起，通过参数注入接收websocket原始数据时，需要加上类型 string。例如： public function echo(string $data)。

//

///3/ 访问

根据以上定义好的 Ws模块、消息解析器、消息控制器 等内容后启动我们的服务。然后打开webscoket 调试工具，链接Ws的地址： ws://localhost:port/chat 然后测试发送一个内容。

Send: testWS

Recv: hi, this is home.index

Send: home.echo:这是数据

Recv: (home.echo)Recv: 这是数据

--

5、消息发送

( http://swoft.org/docs/2.x/zh-CN/websocket-server/message-send.html )

//

上一节我们知道了如何创建ws模块，并通过客户端连接到server。可以从示例代码里看到有简单的消息发送使用了。

...

/\*\* @var \Swoole\WebSocket\Server $server \*/

$server->push($fd, 'hello, welcome! :)');

...

这里的server是swoole的 \Swoole\WebSocket\Server 对象，$fd 是与客户端的连接 ID，它表明了不同的客户端。

//

///1/ 使用 \server()

除了使用 $server 来发送消息外,我们还可以使用swoft封装好的 \server() 或者 \Swoft::server() 来发送消息.

\server()->sendTo($fd, 'hi, 你好啊！');

\Swoft::server()->sendTo($fd, 'hi, 你好啊！');

是 Swoft\WebSocket\Server\WebSocketServer 的实例对象；内部已经封装了各种发送消息的方法API；前台运行时，通过它发送消息能从控制台看到消息发送log。

//

(1)、消息发送API

注意下面的方法都在类：Swoft\WebSocket\Server\WebSocketServer。

//

///1/ 发送给某个客户端

public function sendTo(int $receiver, string $data, int $sender = 0): int

// 参数说明

$receiver int 接收者的fd

$data string 要发送的消息数据

$sender int 发送者的fd。 可选的

// 示例

\server()->sendTo($fd, 'hi, 你好啊！');

//

///2/ 发送给指定的一些客户端

public function sendToSome(string $data, array $receivers = [], array $excluded = [], int $sender = 0, int $pageSize = 50): int

// 参数说明

$data string 要发送的消息数据。

$receivers int[] 指定的接收者fd 列表。

$excluded int[] 排除的接收者fd 列表。

$sender int 发送者的fd，可选的。

// 方法说明

当 $receivers 有数据时，将会忽略 $excluded。 此时就是将消息指定的发给这些接收者。

当 $receivers 为空时，若 $excluded 有值，将会给除了这些人之外的发送消息，若 $excluded 为空，相当于给所有人发消息。

// 示例

\server()->sendToSome('hi, 你们好啊！', [$fd0, $fd1, ...]);

//

///3/ 广播消息

发送消息给除了 sender 外的所有人。使用分页方式发送，每 50 个一页，直到全部发送完毕。

broadcast(string $data, array $receivers = [], array $excluded = [], int $sender = 0): int

//

///4/ 发送给所有客户端

public function sendToAll(string $data, int $sender = 0, int $pageSize = 50): int

发送消息给所有客户端，相当于进行全员广播。使用分页方式发送，每 50 个一页，直到全部发送完毕。

$data string 要发送的消息数据。$sender int 发送者的fd，可选的。

// 示例

\server()->sendToAll('hi, 大家好啊！');

//

///5/ send

// 参数跟 sendToSome 一样

会自动根据参数判断调用上面的（sendTo, sendToAll, sendToSome）中的一个方法。

--

6、异常处理

( http://swoft.org/docs/2.x/zh-CN/websocket-server/exception.html )

前面我们了解了系统如何处理异常，以及http server里如何处理异常的。在websocket server 也是类似的，我们只需定义websocket相关场景的异常处理器就行。

与http server 里只有一个 request 场景不同, websocket 里有四个场景：handshake 握手环节；open 握手后连接打开；message 消息通信阶段；close 连接关闭。

下面我们编写 websocket 几个环节中最重要的 握手 和 消息通信 环节的异常处理。其他环节的可以参考和继承相关类来编写。

//

(1)、握手异常

因为websocket握手环节就是http请求处理，所以此环节的异常跟http里处理是一样的，当然你还是得继承为这个场景设计的基础类才行。

//

///1/ 必须继承 AbstractHandshakeErrorHandler 类，我们才能知道你要处理哪个场景里的异常。

<?php declare(strict\_types=1);

namespace App\Exception\Handler;

use ReflectionException;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Error\Annotation\Mapping\ExceptionHandler;

use Swoft\Http\Message\Response;

use Swoft\WebSocket\Server\Exception\Handler\AbstractHandshakeErrorHandler;

use Throwable;

use function get\_class;

use function sprintf;

use const APP\_DEBUG;

/\*\*

 \* Class HttpExceptionHandler

 \*

 \* @ExceptionHandler(\Throwable::class)

 \*/

class WsHandshakeExceptionHandler extends AbstractHandshakeErrorHandler

{

    /\*\*

     \* @param Throwable $e

     \* @param Response  $response

     \*

     \* @return Response

     \* @throws ReflectionException

     \* @throws ContainerException

     \*/

    public function handle(Throwable $e, Response $response): Response

    {

        // Debug is false

        if (!APP\_DEBUG) {

            return $response->withStatus(500)->withContent(sprintf(

                '%s At %s line %d', $e->getMessage(), $e->getFile(), $e->getLine()

            ));

        }

        $data = [

            'code'  => $e->getCode(),

            'error' => sprintf('(%s) %s', get\_class($e), $e->getMessage()),

            'file'  => sprintf('At %s line %d', $e->getFile(), $e->getLine()),

            'trace' => $e->getTraceAsString(),

        ];

        // Debug is true

        return $response->withData($data);

    }

}

?>

//

(2)、消息通信异常

在握手成功后的消息通信阶段出现异常，也可以方便的捕获处理。注意：你仍然需要继承专有场景的异常处理抽象类 AbstractMessageErrorHandler。

<?php declare(strict\_types=1);

namespace App\Exception\Handler;

use ReflectionException;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Error\Annotation\Mapping\ExceptionHandler;

use Swoft\Log\Helper\Log;

use Swoft\WebSocket\Server\Exception\Handler\AbstractMessageErrorHandler;

use Swoole\WebSocket\Frame;

use Throwable;

use function server;

use const APP\_DEBUG;

/\*\*

 \* Class WsMessageExceptionHandler

 \*

 \* @since 2.0

 \*

 \* @ExceptionHandler(\Throwable::class)

 \*/

class WsMessageExceptionHandler extends AbstractMessageErrorHandler

{

    /\*\*

     \* @param Throwable $e

     \* @param Frame     $frame

     \*

     \* @throws ContainerException

     \* @throws ReflectionException

     \*/

    public function handle(Throwable $e, Frame $frame): void

    {

        $message = sprintf('%s At %s line %d', $e->getMessage(), $e->getFile(), $e->getLine());

        Log::error('Ws server error(%s)', $message);

        // Debug is false

        if (!APP\_DEBUG) {

            server()->push($frame->fd, $e->getMessage());

            return;

        }

        server()->push($frame->fd, $message);

    }

}

?>

--

--

四、RPC Server

( http://swoft.org/docs/2.x/zh-CN/rpc-server/index.html )

--

0、RPC 服务

( http://swoft.org/docs/2.x/zh-CN/rpc-server/index.html )

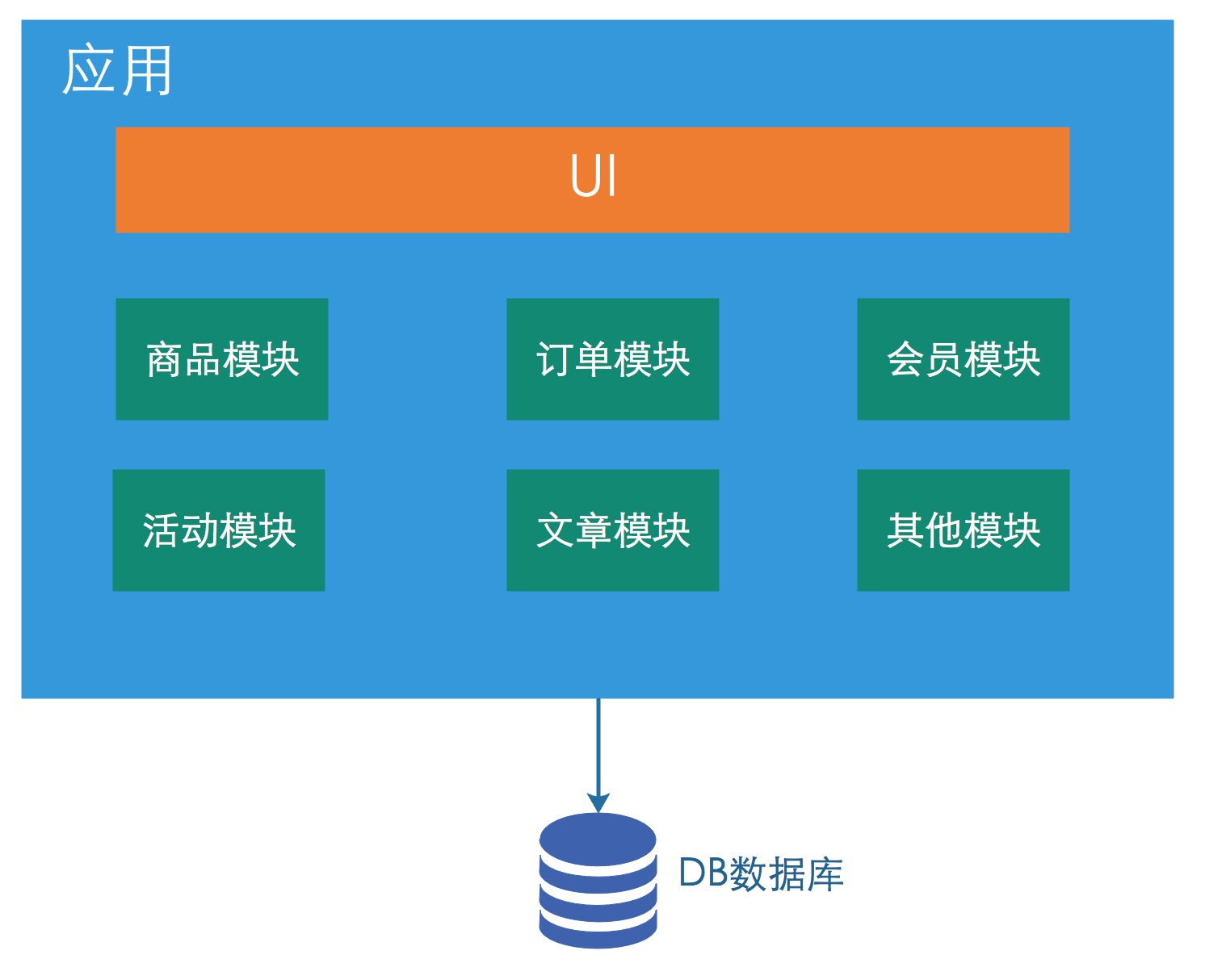
RPC，是一种远程调用方式（Remote Procedure Call），通过RPC我们可以像调用本地方法一样调用别的机器上的方法，用户将无感服务器与服务器之间的通讯。RPC在微服务当中起到相当大的作用，当然RPC不是微服务必须的一种方式，有别的方式也可以实现这种远程调用例如RESTful API就可以实现远程调用。如果有用过SOAP那么你使用RPC将会觉得很类似，都是可以直接调用别的机器上的方法。

随着业务的发展我们的项目从简单的单体结构逐渐的演化成微服务结构，我们为什么要拆分成微服务呢？那我们来说说微服务和单体架构的优缺点。我们看一下单体架构图。

//

(1)、单体架构

( 单体架构图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/rpc-server/1.png )



//

///1/ 单体架构优点

部署容易，如php写的项目，只要一个文件夹复制到支持php的环境就可以了，java只需要一个jar包。

测试容易，我们整体项目只要改了一个地方马上就可以测试得出结果。

负载均衡就可以解决，快速部署多个一模一样的项目在不同的机器运行分流。

//

///2/ 单体架构缺点

部署的问题，对于php来说这点还好，但是对于java的项目来说，我们需要重新打包整个项目耗费的时间是很长的。

代码维护，由于所有的代码都写在一个项目里面，要想要修改某一个功能点那么需要对项目的整体逻辑和设计有较深的理解，否则代码耦合严重，导致维护难，特别对于新入职的员工来说这将是最容易出现问题的地方。

开发效率低，随着项目需求的不断改变和新的功能新增，老旧的代码又不敢随便删除，导致整个项目变得笨重，这将会增加你阅读代码的时间。

扩展性，在高并发的情况下，我们往往不是整个项目的每一个功能都处于高流量高请求的情况下的，很多时候都是某一个功能模块使用的人数比较多，在单体结构下我们没有办法针对单个功能实现分布式扩展，必须整个项目一起部署。

//

(2)、微服务架构

在2014年被提出，现在国内很多公司已经使用，微服务是一种架构设计，并不是说什么框架或者代替什么。微服务做的事情是按照项目颗粒度进行服务的拆分，把模块单独拿出来做成每一个单独的小项目。微服务的主要特点有：每一个功能模块是一个小项目、独立运行在不同进程或者机器上、不同功能可以由不同的人员开发独立开发不松耦合、独立部署不需要依赖整体项目就可以启动单个服务、分布式管理。每一个服务只要做好自己的事情就好了。在设计微服务的时候还需要考虑到数据库的问题，是所有微服务使用共同一个数据库还是每一个服务单个数据库。

//

///1/ 微服务优点

|1>、拆分业务，把整体大项目分割成不同小项目运行在不同进程或者机器上实现数据隔离。

|2>、技术栈，每个服务可以由不同的团队或者开发者进行开发，外部调用人员不需要操心具体怎么实现的，只需要类似调用自己方法一样或者接口一样按照服务提供者给出来的参数传递即可。

|3>、独立部署，每一个服务独立部署，部署一个服务不会影响整体项目，如果部署失败最多是这个服务的功能缺失，并不影响其他功能的使用。

|4>、按需部署，针对不同的需求可以给不同的服务自由扩展服务器，根据服务的规模部署满足需求的实例。

|5>、局部修改，当一个服务有新需求或者其他修改，不需要修改整体项目只要管好自己的服务就好了。

//

///2/ 微服务缺点

|1>、运维，微服务由于把业务拆分得细，有可能部署在不同机器上，因此对于运维人员的管理来说，这部分的成本会加大。

|2>、接口调整，微服务之间通过接口进行通信。如果修改某个微服务的API，可能所有使用了该接口的微服务都需要做调整。

接口调整，微服务之间通过接口进行通信。如果修改某个微服务的API，可能所有使用了该接口的微服务都需要做调整；

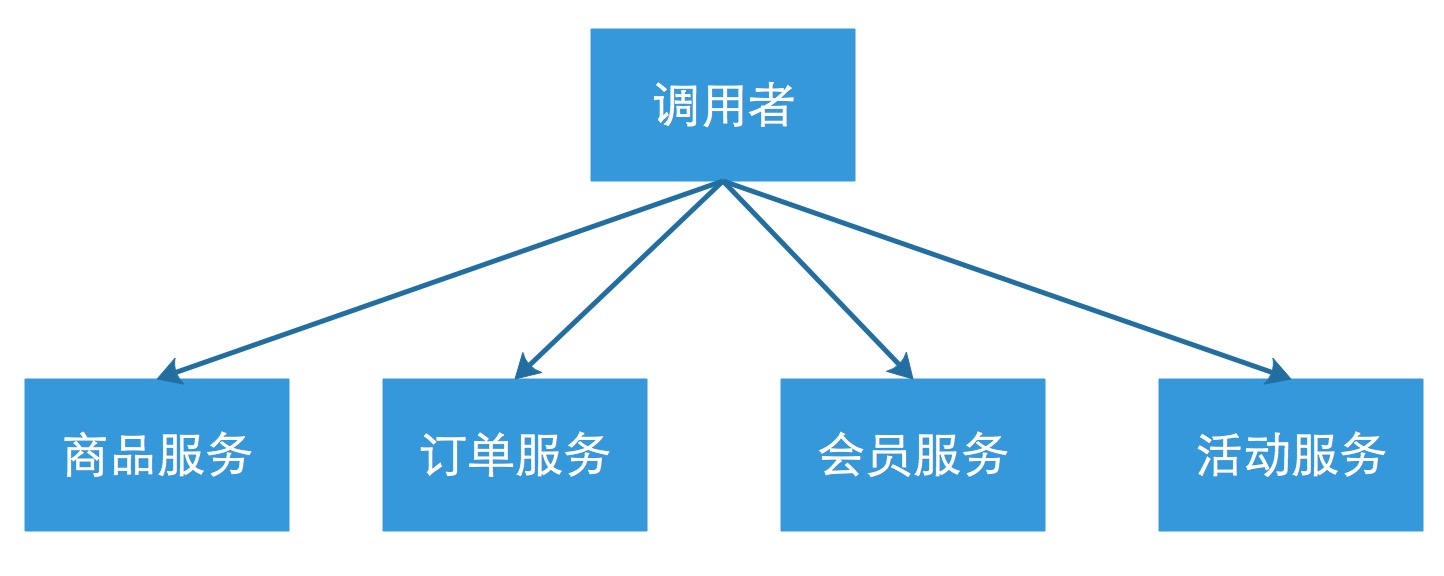
|3>、重复劳动，很多服务可能都会使用到相同的功能。而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个服务都会开发这一功能，导致代码重复。

|4>、分布式，由于会把不同服务部署在不同机器上，那么对于这些服务的调用、容错、网络延迟、分布式事务等等都是一个很大的挑战，当然微服务不一定全部都是部署在不同服务器上

//

(3)、服务调用

( 微服务调用图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/rpc-server/2.png )



如上图所示，RPC就用于调用者与服务之间的通讯，RPC协议可基于TCP、UDP或者HTTP实现，但是更推荐选择TCP。

//

///1/ RPC或者RESTfulAPI调用区别

/// 例如调用者需要调用商品的服务就可以通过RPC或者RESTful API来调用，那么RPC调用和RESTful API两者之间的区别在哪呢？

|1>、TCP 支持长连接，当调用服务的时候不需要每次都进行三次握手才实现。从性能和网络消耗来说RPC都具备了很好的优势。

|2>、RESTful API 基于HTTP的，也就是说每次调用服务都需要进行三次握手建立起通信才可以实现调用，当我们的并发量高的时候这就会浪费很多带宽资源。

|3>、服务对外的话采用RESTful API会比RPC更具备优势，因此看自己团队的服务是对内还是对外。

//

///2/ RPC调用过程

( RPC调用过程图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/rpc-server/3.png )



提示：RPC最主要的作用就是用于服务调用。

本文作为RPC的使用场景开山篇，对于单体架构和微服务的进行了一个描述。这个就是RPC的一个使用场景，也是最常用的一个使用场景。大家只有了解好RPC是什么使用在什么场景才能更好的去使用。

Swoft给我们提供了RPC的底层服务，我们并不需要去关心底层通讯细节和调用的过程。Swoft通过定义接口，实现接口，启动RPC Server 提供接口服务。我们只需要简单的写好几个类就可以实现一个简单RPC模块。

--

1、常用命令

( http://swoft.org/docs/2.x/zh-CN/rpc-server/command.html )

//

///1/ RPC命令查看：在项目根目录执行如下命令；

$ php bin/swoft rpc

Usage:

  bin/swoft rpc:COMMAND [--opt ...] [arg ...]

Global Options:

      --debug      Setting the application runtime debug level(0 - 4)

      --no-color   Disable color/ANSI for message output

  -h, --help       Display this help message

  -V, --version    Show application version information

Commands:

  reload      Reload worker processes

  restart     Restart the http server

  start       Start the http server

  stop        Stop the currently running server

Example:

 bin/swoft rpc:start     Start the rpc server

 bin/swoft rpc:stop      Stop the rpc server

//

///2/ Rpc 的命令都在 Commands

reload 重新加载 worker 进程。

restart 重启 RPC 服务器。

start 启动 RPC 服务器。

stop 停止 RPC 服务器。

//

(1)、使用

///1/ 前台运行

$ php bin/swoft rpc:start

                            Information Panel

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* RPC      | Listen: 0.0.0.0:18307, type: TCP, mode: Process, worker: 1

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RPC server start success !

//

///2/ 后台运行

$ php bin/swoft rpc:start -d

                            Information Panel

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* RPC      | Listen: 0.0.0.0:18307, type: TCP, mode: Process, worker: 1

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RPC server start success !

--

2、配置参数

( http://swoft.org/docs/2.x/zh-CN/rpc-server/setting.html )

RPC 服务启动有单独启动和集成其它服务(Http/Websocket)两种方式，无论哪种方式都首先要在 bean.php 配置RPC。

return [

    'rpcServer'  => [

        'class' => ServiceServer::class,

        'port' => 18308,

    ],

]

port 配置启动端口号；setting 启动配置参数，对应 swooleServer->setting。

//

///1/ Http server 启动中集成 RPC 服务

return [

    'httpServer' => [

        'class'    => HttpServer::class,

        'port'     => 18306,

        'listener' => [

            'rpc' => bean('rpcServer')

        ],

        // ...

    ],

]

listener 单独监听一个RPC服务，且同时可以监听多个 RPC 服务。如果是单独启动，无效其它配置直接可以启动。

--

3、声明服务

( http://swoft.org/docs/2.x/zh-CN/rpc-server/statement.html )

(0)、接口服务

定义接口并实现接口，才能提供RPC服务。

//

(1)、目录定义

///1/ 官方应用中给出的目录如下

app/

  Rpc/

    - Lib/          // 服务的公共接口定义目录，里面通常只有php接口类

    - Services/     // 具体的服务接口实现类，里面的类通常实现了 Lib 中定义的接口

当然在多个服务中使用时， 要将lib库 app/Rpc/Lib 移到一个公共的git仓库里，然后各个服务通过 composer 来获取使用。

//

(2)、定义接口

服务提供方定义好接口格式，存放到公共的lib库里面，服务调用方，加载lib库，就能使用接口服务，接口定义和普通接口完全一致。

\*\*

 \* Class UserInterface

 \*

 \* @since 2.0

 \*/

interface UserInterface

{

    /\*\*

     \* @param int   $id

     \* @param mixed $type

     \* @param int   $count

     \*

     \* @return array

     \*/

    public function getList(int $id, $type, int $count = 10): array;

    /\*\*

     \* @param int $id

     \*

     \* @return bool

     \*/

    public function delete(int $id): bool;

    /\*\*

     \* @return string

     \*/

    public function getBigContent(): string;

}

//

(3)、接口实现

一个接口，会存在多种不同的实现，通过一个版本号来标识是哪个逻辑实现。

///1/ 注解

@Service

version 定义接口版本，默认是 1.0。

//

///2/ 实例

/// 实现版本1

/\*\*

 \* Class UserService

 \*

 \* @since 2.0

 \*

 \* @Service()

 \*/

class UserService implements UserInterface

{

    /\*\*

     \* @param int   $id

     \* @param mixed $type

     \* @param int   $count

     \*

     \* @return array

     \*/

    public function getList(int $id, $type, int $count = 10): array

    {

        return ['name' => ['list']];

    }

    /\*\*

     \* @param int $id

     \*

     \* @return bool

     \*/

    public function delete(int $id): bool

    {

        return false;

    }

    /\*\*

     \* @return string

     \*/

    public function getBigContent(): string

    {

        $content = Co::readFile(\_\_DIR\_\_ . '/big.data');

        return $content;

    }

}

//

/// 实现版本2

/\*\*

 \* Class UserServiceV2

 \*

 \* @since 2.0

 \*

 \* @Service(version="1.2")

 \*/

class UserServiceV2 implements UserInterface

{

    /\*\*

     \* @param int   $id

     \* @param mixed $type

     \* @param int   $count

     \*

     \* @return array

     \*/

    public function getList(int $id, $type, int $count = 10): array

    {

        return [

            'name' => ['list'],

            'v'    => '1.2'

        ];

    }

    /\*\*

     \* @param int $id

     \*

     \* @return bool

     \*/

    public function delete(int $id): bool

    {

        return false;

    }

    /\*\*

     \* @return string

     \*/

    public function getBigContent(): string

    {

        $content = Co::readFile(\_\_DIR\_\_ . '/big.data');

        return $content;

    }

}

不同的实现，需要定义不同的唯一版本号，如果存在相同，加载之后的服务会覆盖之前的服务。

--

--

五、RPC Client

( http://swoft.org/docs/2.x/zh-CN/rpc-client/index.html )

服务调用方法，通过使用服务提供方法，提供的lib接口，调用接口实现服务，不需要了解实现细节。

//

1、配置服务

return [

    'user'       => [

        'class'   => ServiceClient::class,

        'host'    => '127.0.0.1',

        'port'    => 18307,

        'setting' => [

            'timeout'         => 0.5,

            'connect\_timeout' => 1.0,

            'write\_timeout'   => 10.0,

            'read\_timeout'    => 0.5,

        ],

        'packet'  => bean('rpcClientPacket')

    ],

    'user.pool'  => [

        'class'  => ServicePool::class,

        'client' => bean('user')

    ],

];

如上定义了一个 user 服务，连接池配置参数和其它一样。

//

2、如何使用

( http://swoft.org/docs/2.x/zh-CN/rpc-client/usage.html )

/\*\*

 \* Class RpcController

 \*

 \* @since 2.0

 \*

 \* @Controller()

 \*/

class RpcController

{

    /\*\*

     \* @Reference(pool="user.pool")

     \*

     \* @var UserInterface

     \*/

    private $userService;

    /\*\*

     \* @Reference(pool="user.pool", version="1.2")

     \*

     \* @var UserInterface

     \*/

    private $userService2;

    /\*\*

     \* @RequestMapping("getList")

     \*

     \* @return array

     \*/

    public function getList(): array

    {

        $result  = $this->userService->getList(12, 'type');

        $result2 = $this->userService2->getList(12, 'type');

        return [$result, $result2];

    }

    /\*\*

     \* @RequestMapping("returnBool")

     \*

     \* @return array

     \*/

    public function returnBool(): array

    {

        $result = $this->userService->delete(12);

        if (is\_bool($result)) {

            return ['bool'];

        }

        return ['notBool'];

    }

    /\*\*

     \* @RequestMapping()

     \*

     \* @return array

     \*/

    public function bigString(): array

    {

        $string = $this->userService->getBigContent();

        return ['string'];

    }

}

//

(1)、@Reference

pool 指定使用那个服务的连接池(使用那个服务)。

version 指定服务的版本。

//

(2)、非 Swoft 框架调用

默认消息协议是 json-rpc， 所以我们按照这个格式就可以了，需要注意的是，默认消息协议是以 \r\n\r\n 结尾的。

这里 method 的格式为 "{version}::{class\_name}::{method\_name}"。

{

    "jsonrpc": "2.0",

    "method": "{version}::{class\_name}::{method\_name}",

    "params": [],

    "id": "",

    "ext": []

}

///1/ 代码示例

// 如果使用默认消息协议，可以按照如下方式进行封装

<?php

const RPC\_EOL = "\r\n\r\n";

function request($host, $class, $method, $param, $version = '1.0', $ext = []) {

    $fp = stream\_socket\_client($host, $errno, $errstr);

    if (!$fp) {

        throw new Exception("stream\_socket\_client fail errno={$errno} errstr={$errstr}");

    }

    $req = [

        "jsonrpc" => '2.0',

        "method" => sprintf("%s::%s::%s", $version, $class, $method),

        'params' => $param,

        'id' => '',

        'ext' => $ext,

    ];

    $data = json\_encode($req) . RPC\_EOL;

    fwrite($fp, $data);

    $result = '';

    while (!feof($fp)) {

        $tmp = stream\_socket\_recvfrom($fp, 1024);

        if ($pos = strpos($tmp, RPC\_EOL)) {

            $result .= substr($tmp, 0, $pos);

            break;

        } else {

            $result .= $tmp;

        }

    }

    fclose($fp);

    return json\_decode($result, true);

}

$ret = request('tcp://127.0.0.1:18307', \App\Rpc\Lib\UserInterface::class, 'getList',  [1, 2], "1.0");

var\_dump($ret);

?>

//

3、1.0 RPC

( http://swoft.org/docs/2.x/zh-CN/rpc-client/rpc-1.0.html )

如果系统之前使用的是 Swoft 1.0 RPC server，Swoft 2.0 定义了一种兼容 1.0 RPC 协议，使用很简单。(Available: >= v2.0.3)

//

(1)、配置

使用 2.0 框架中调用 1.0 RPC server 提供的服务，首先必须配置(app/bean.php) 1.0 RPC 协议。

return [

    // ...

    'user'             => [

        'class'   => ServiceClient::class,

        'host'    => '127.0.0.1',

        'port'    => 8099,

        'setting' => [

            'timeout'         => 0.5,

            'connect\_timeout' => 1.0,

            'write\_timeout'   => 10.0,

            'read\_timeout'    => 0.5,

            'package\_eof'     => "\r\n",

        ],

        'packet'  => bean('rpcClientSwoftPacketV1')

    ],

    // ...

];

host/port 配置 1.0 地址和端口即可。package\_eof 必须配置数据包结尾符，1.0 包结尾符是 \r\n。packet 必须配置使用 bean('rpcClientSwoftPacketV1') 1.0 打包器。

//

(2)、使用

以上配置完成后，就可以直接使用了。这里直接以调用 Swoft 1.x 的 App\Lib\DemoInterface 为例。

/\*\*

 \* Class RpcController

 \*

 \* @since 2.0

 \*

 \* @Controller()

 \*/

class RpcController

{

    /\*\*

     \* @Reference(pool="user.pool", version="0")

     \*

     \* @var DemoInterface

     \*/

    private $demoServcie;

    /\*\*

     \* @RequestMapping(route="swoftV1")

     \*

     \* @return array

     \*/

    public function swoftV1():array {

        return [$this->demoServcie->getUser('1')];

    }

}

调用 1.x RPC version 必须指定，因为 2.x 与 1.0 默认值不一样。

不能调用 1.x 的 deferXxxx 方法 2.0 已经丢弃。

2.x 里面调用的接口必须和 1.x 接口命名空间、类名以及方法名称参数完全一样。

--

--

六、TCP Server

( http://swoft.org/docs/2.x/zh-CN/tcp-server/index.html )

0、Tcp 服务器

( http://swoft.org/docs/2.x/zh-CN/tcp-server/index.html )

从 swoft 2.0.4 版本开始，提供经过swoft封装的tcp 服务器实现。在原有swoole server的基础上，封装并细化功能使用。(Available: >= v2.0.4)

//

(1)、功能特色

基于swoft的注解系统，使用方便快速。

提供统一的协议设置，同时支持EOF和length两种切包方式。

完善的数据收发解析，统一的上下文/请求与响应对象封装。

内置请求调度处理，可以像http一样细致的分发请求数据到不同的方法处理。

内置支持多种打包方式(json php token)，同时可以自由扩展。

//

(2)、安装

composer require swoft/tcp-server

//

(3)、Git仓库

tcp 数据协议 https://github.com/swoft-cloud/swoft-tcp

tcp-server https://github.com/swoft-cloud/swoft-tcp-server

//

(4)、参与贡献

欢迎参与贡献，您可以：fork 我们的开发仓库 swoft/component；修改代码然后发起 PR；关于发起PR的注意事项。

--

1、配置服务

(0)、tcp server 配置

tcp server 的 host, port 等配置是都是完全可以自定义的。 配置需要编辑 app/bean.php 文件，下面列举了一些简单的配置，你也可以自由组合同时提供多种服务。tcp server 的默认端口是 18309。

//

(1)、可配置项

///1/ 可配置项用于 tcpServer bean 配置，除了 class 其他都是 TcpServer 的属性。

class 指定 tcp server 的bean类，默认即是 Swoft\Tcp\Server\TcpServer::class。

port 指定 tcp server 的端口。

listener 指定其他一同启动的服务，添加端口服务监听，可以多个，rpc 启动 RPC 服务。

on 配置监听的事件，注册swoole事件、设置对应事件的处理监听。

setting 这里是参考 Swoole Server配置选项( https://wiki.swoole.com/wiki/page/274.html )。

pidFile 设置进程 pid文件 位置，默认值 @runtime/swoft-tcp.pid。

mode 运行的模式，参考 Swoole Server 构造函数( https://wiki.swoole.com/wiki/page/14.html ) 第三个参数。

type 指定Socket的类型，支持TCP、UDP、TCP6、UDP6、UnixSocket Stream/Dgram 等 Swoole Server 构造函数( https://wiki.swoole.com/wiki/page/14.html ) 第四个参数。

//

(2)、基础配置

// ...

'tcpServer'   => [

    'class'   => TcpServer::class,

    'port' => 18309,

    'debug' => env('SWOFT\_DEBUG', 0),

    /\* @see TcpServer::$setting \*/

    'setting' => [

        'log\_file' => alias('@runtime/swoole.log'),

    ],

],

/\*\* @see \Swoft\Tcp\Protocol \*/

'tcpServerProtocol' => [

    'type'            => \Swoft\Tcp\Packer\SimpleTokenPacker::TYPE,

    // 'openEofCheck'    => true, // Defalut use EOF check

    // 'openLengthCheck' => true,

],

//

///1/ 协议配置

通常你只需配置好协议的分包方式，内部的细节配置会自动同步设置到TcpServer。

/\*\* @see \Swoft\Tcp\Protocol \*/

'tcpServerProtocol' => [

    'type'            => \Swoft\Tcp\Packer\SimpleTokenPacker::TYPE,

    // 'openEofCheck'    => true, // Defalut use EOF check

    // 'openLengthCheck' => true,

],

//

//// 可配置项

type string 默认的数据打包器的类型，默认是 token-text。

packers array 可用的数据打包器的列表，内置了 json php token-text 三种。

packageMaxLength int 同 swoole 的 package\_max\_length，默认 81920。

openEofCheck bool 同 swoole 的 open\_eof\_check 默认 true。

openLengthCheck bool 同 swoole 的 open\_length\_check，总是与 openEofCheck 相反。默认 false。

//

(3)、添加RPC服务

如果你想运行tcp server时，同时启动RPC Server服务。

// ...

'tcpServer'   => [

    'listener' => [

        'rpc' => \bean('rpcServer') // 引入 rpcServer

    ],

],

'rpcServer'  => [

    'class' => ServiceServer::class,

    'port' => 18308,

],

//

(4)、启用全部功能

task process；

rpc server；

// ...

'tcpServer'   => [

    'class'   => TcpServer::class,

    'port' => 18307,

    'on'      => [

        // 启用任务必须添加 task, finish 事件处理

        SwooleEvent::TASK   => bean(TaskListener::class),

        SwooleEvent::FINISH => bean(FinishListener::class)

    ],

    'listener' => [

        // 引入 rpcServer

        'rpc' => \bean('rpcServer')

    ],

    'debug' => env('SWOFT\_DEBUG', 0),

    /\* @see TcpServer::$setting \*/

    'setting' => [

        'log\_file' => alias('@runtime/swoole.log'),

        // 任务需要配置 task worker

        'task\_worker\_num'       => 2,

        'task\_enable\_coroutine' => true

    ],

],

'rpcServer'  => [

    'class' => ServiceServer::class,

    'port' => 18308,

],

ok, 现在通过 php bin/swoft tcp:start 启动的服务器，就支持上面的全部功能了。

--

2、管理服务

( http://swoft.org/docs/2.x/zh-CN/tcp-server/manage.html )

(1)、启动与管理

在项目目录下执行如下命令可以看到tcp server的管理命令，跟http server的管理命令一致。

$ php bin/swoft tcp

Description:

  There some commands for manage the tcp server

Usage:

  tcp:{command} [arguments] [options]

Commands:

  start    Start the tcp server

  stop     Stop the running server

  restart  Restart the running server

Options:

  -h, --help  Show help of the command group or specified command action

//

(2)、启动

///1/ 前台运行

$ php bin/swoft tcp:start

Notice: tcp server default port is 18309

//

///2/ 后台运行

$ php bin/swoft tcp:start -d

//

///3/ 重新启动

$ php bin/swoft tcp:restart

--

3、事件通知 (Tcp 事件通知)

( http://swoft.org/docs/2.x/zh-CN/tcp-server/event.html )

通常情况下，我们无需关心tcp server 相关的 connect close 事件。 但 swoft 内部都是监听并触发了框架内部定义的相关事件，你同样可以监听并处理一些逻辑。

(1)、事件列表

<?php declare(strict\_types=1);

namespace Swoft\Tcp\Server;

/\*\*

 \* Class TcpServerEvent

 \*

 \* @since 2.0

 \*/

final class TcpServerEvent

{

    /\*\*

     \* On connect

     \*/

    public const CONNECT = 'swoft.tcp.server.connect';

    /\*\*

     \* On connect error

     \*/

    public const CONNECT\_ERROR = 'swoft.tcp.server.connect.error';

    /\*\*

     \* On receive

     \*/

    public const RECEIVE = 'swoft.tcp.server.receive';

    /\*\*

     \* On receive error

     \*/

    public const RECEIVE\_ERROR = 'swoft.tcp.server.receive.error';

    /\*\*

     \* On close

     \*/

    public const CLOSE = 'swoft.tcp.server.close';

    /\*\*

     \* On close error

     \*/

    public const CLOSE\_ERROR = 'swoft.tcp.server.close.error';

}

?>

//

(2)、监听事件

跟其他事件一样，直接通过 @Lisenter 监听对应事件名，就可以处理相关逻辑了。

<?php declare(strict\_types=1);

namespace App\Listener;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Tcp\Server\TcpServerEvent;

/\*\*

 \* Class UserSavingListener

 \*

 \* @since 2.0

 \*

 \* @Listener(TcpServerEvent::CONNECT)

 \*/

class TcpConnectListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        /\* @var \Swoole\Server $server \*/

        $server = $event->getTarget();

        var\_dump(

          $event->getParam(0), // fd

          $event->getParam(1), // reactorId

        );

    }

}

?>

--

4、控制器 (TCP控制器)

( http://swoft.org/docs/2.x/zh-CN/tcp-server/controller.html )

与http server类似，tcp server中也使用对应的控制器来处理系统分发的数据请求。

//

(1)、注解

tcp server新增两个注解 @TcpControler 和 @TcpMapping，由他们定义tcp控制器和处理方法。

//

///1/ TcpControler

类注解 @TcpControler 标记当前类是一个 Tcp 控制器。

注解类：Swoft\Tcp\Server\Annotation\Mapping\TcpController。

作用范围：CLASS。

拥有属性：prefix string 数据路由前缀，为空自动解析类名称为前缀。

//

///2/ TcpMapping

方法注解 @TcpMapping 标记具体的数据处理方法，类似于http控制器里的action。

注解类：Swoft\Tcp\Server\Annotation\Mapping\TcpMapping。

作用范围：METHOD。

// 拥有属性

route string 命令名称，为空自动使用方法名称。

root bool 命令名称是否是顶级命令，默认 false。

//

///3/ 提示

自动解析 TcpControler 的前缀时，会自动尝试去除 Controler 部分。eg: DemoController 得到 demo。

通常，完整的tcp命令是 上面的 prefix 和 route 由点拼接而成 PREFIX.ROUTE，eg: demo.index。

当 TcpMapping.root 为 true 时，完整命令直接是 TcpMapping.route。

//

(2)、编写控制器

<?php declare(strict\_types=1);

namespace App\Tcp\Controller;

use Swoft\Tcp\Server\Annotation\Mapping\TcpController;

use Swoft\Tcp\Server\Annotation\Mapping\TcpMapping;

use Swoft\Tcp\Server\Request;

use Swoft\Tcp\Server\Response;

/\*\*

 \* Class DemoController

 \*

 \* @TcpController()

 \*/

class DemoController

{

    /\*\*

     \* @TcpMapping("list", root=true)

     \* @param Response $response

     \*/

    public function list(Response $response): void

    {

        $response->setData('[list]allow command: list, echo, demo.echo');

    }

    /\*\*

     \* @TcpMapping("echo")

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function index(Request $request, Response $response): void

    {

        $str = $request->getPackage()->getDataString();

        $response->setData('[demo.echo]hi, we received your message: ' . $str);

    }

    /\*\*

     \* @TcpMapping("strrev", root=true)

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function strRev(Request $request, Response $response): void

    {

        $str = $request->getPackage()->getDataString();

        $response->setData(\strrev($str));

    }

    /\*\*

     \* @TcpMapping("echo", root=true)

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function echo(Request $request, Response $response): void

    {

        $str = $request->getPackage()->getDataString();

        $response->setData('[echo]hi, we received your message: ' . $str);

    }

}

?>

好了，服务端代码已经编写好了。这里我们使用默认的配置 EOF 分包方式，数据协议格式也使用默认的 SimpleTokenPacker::TYPE。重新启动我们的 tcp server php bin/swoft tcp:start，接下来一篇讲述如何与我们的tcp server进行通信交互。

--

5、客户端通信

( http://swoft.org/docs/2.x/zh-CN/tcp-server/client-communicate.html )

你可以直接使用 swoole 提供的 Swoole\Coroutine\Client 作为tcp客户端，快速的对接swoft的tcp sever。

//

///1/ 为了分包和数据解析与tcp server保持一致，你需要依赖tcp协议包

composer require swoft/tcp

开始之前，首先你得确认你已经启动了tcp server端，并且保持客户端与服务端的 协议设置是一致的。

//

(1)、swoft 示例

<?php declare(strict\_types=1);

namespace App\Command;

use Swoft\Tcp\Protocol;

use Swoole\Coroutine\Client;

use Swoft\Console\Helper\Show;

use Swoft\Console\Input\Input;

use Swoft\Console\Output\Output;

use const SWOOLE\_SOCK\_TCP;

// ...

    public function tcpTest(Input $input, Output $output): void

    {

        $proto = new Protocol();

        // If your tcp server use length check.

        // $proto->setOpenLengthCheck(true);

        var\_dump($proto->getConfig());

        $host = '127.0.0.1';

        $port = 18309;

        $client = new Client(SWOOLE\_SOCK\_TCP);

        // Notice: config client

        $client->set($proto->getConfig());

        if (!$client->connect((string)$host, (int)$port, 5.0)) {

            $code = $client->errCode;

            /\*\* @noinspection PhpComposerExtensionStubsInspection \*/

            $msg = socket\_strerror($code);

            $output->error("Connect server failed. Error($code): $msg");

            return;

        }

        // Send message $msg . $proto->getPackageEOf()

        if (false === $client->send($proto->packBody($msg))) {

            /\*\* @noinspection PhpComposerExtensionStubsInspection \*/

            $output->error('Send error - ' . socket\_strerror($client->errCode));

            return;

        }

        // Recv response

        $res = $client->recv(2.0);

        if ($res === false) {

            /\*\* @noinspection PhpComposerExtensionStubsInspection \*/

            $output->error('Recv error - ' . socket\_strerror($client->errCode));

            return;

        }

        if ($res === '') {

            $output->info('Server closed connection');

            return;

        }

        // unpack response data

        [$head, $body] = $proto->unpackData($res);

        $output->prettyJSON($head);

        $output->writef('<yellow>server</yellow>> %s', $body);

    }

// ...

?>

//

(2)、非swoft示例

注意：这里使用的json数据，因此你需要将服务端 tcpServerProtocol 的 type 配置为 json。

<?php

const PKG\_EOF = "\r\n\r\n";

function request(string $host, string $cmd, $data, $ext = []) {

    $fp = stream\_socket\_client($host, $errno, $errstr);

    if (!$fp) {

        throw new Exception("stream\_socket\_client fail errno={$errno} errstr={$errstr}");

    }

    $req = [

        'cmd'  => $cmd,

        'data' => $data,

        'ext' => $ext,

    ];

    $data = json\_encode($req) . PKG\_EOF;

    fwrite($fp, $data);

    $result = '';

    while (!feof($fp)) {

        $tmp = stream\_socket\_recvfrom($fp, 1024);

        if ($pos = strpos($tmp, PKG\_EOF)) {

            $result .= substr($tmp, 0, $pos);

            break;

        } else {

            $result .= $tmp;

        }

    }

    fclose($fp);

    return json\_decode($result, true);

}

$ret = request('tcp://127.0.0.1:18309', 'echo', 'i an client');

var\_dump($ret);

?>

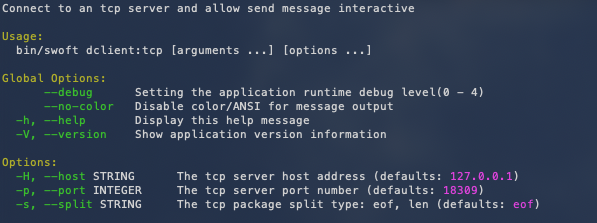
//

(3)、测试通信

你可以复制上面的示例代码，新建一个php文件来运行测试。当然，最方便直接的就是使用我们 devtool 包里提供的 dclient:tcp 工具命令。

运行：php bin/swoft dclient:tcp -h 查看命令帮助。

( 效果图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tcp-server/devtool-tcp-test.png )



//

///1/ php bin/swoft dclient:tcp 测试Tcp服务

//

//0/ 服务控制器方法

<?php declare(strict\_types=1);

/\*\*

 \* ^2\_3^Tcp服务端 (处理系统分发数据请求(数据分包|数据解析))

 \* This file is part of Swoft.

 \*

 \* @link     https://swoft.org

 \* @document https://swoft.org/docs

 \* @contact  group@swoft.org

 \* @license  https://github.com/swoft-cloud/swoft/blob/master/LICENSE

 \*/

namespace App\Tcp\Controller;

use Swoft\Tcp\Server\Annotation\Mapping\TcpController;

use Swoft\Tcp\Server\Annotation\Mapping\TcpMapping;

use Swoft\Tcp\Server\Request;

use Swoft\Tcp\Server\Response;

use function strrev;

/\*\*

 \* Class DemoController

 \*

 \* [Tcp服务]\_Demo控制器;

 \*

 \* @TcpController()

 \*/

class DemoController

{

    /\*\*

     \* @TcpMapping("list", root=true)

     \* @param Response $response

     \*/

    public function list(Response $response): void

    {

        $response->setData('[list]allow command: list, echo, demo.echo');

    }

    /\*\*

     \* @TcpMapping("echo")

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function index(Request $request, Response $response): void

    {

        $str = $request->getPackage()->getDataString();

        $response->setData('[demo.echo]hi, we received your message: ' . $str);

    }

    /\*\*

     \* @TcpMapping("strrev", root=true)

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function strRev(Request $request, Response $response): void

    {

        $str = $request->getPackage()->getDataString();

        $response->setData(strrev($str));

    }

    /\*\*

     \* @TcpMapping("echo", root=true)

     \* @param Request  $request

     \* @param Response $response

     \*/

    public function echo(Request $request, Response $response): void

    {

        // $str = $request->getRawData();

        $str = $request->getPackage()->getDataString();

        $response->setData('[echo]hi, we received your message: ' . $str);

    }

}

?>

//

//1/ 进入客户端

$ php bin/swoft dclient:tcp

//

//2/ 执行Tcp服务方法

// echo 为Tcp服务方法(空格后面就是要发送的数据)；

$ client> echo 客服端发送的消息。

head: {"type":"token-text"}

server> [echo]hi, we received your message: 客服端发送的消息。

$ client>

--

--

七、进程

( http://swoft.org/docs/2.x/zh-CN/process/index.html )

//

0、进程概述

Swoft 进程组件包括如下三部分功能：进程操作；自定义用户进程；进程池。( 2.0.4+ 支持 )

//

(1)、安装

// 使用进程组件之前，先安装组件

composer require swoft/process

//

(2)、使用

安装成功后，不同的功能有不同的使用方式，详解其对应章节。

--

1、进程

( http://swoft.org/docs/2.x/zh-CN/process/process.html )

//

Swoft 框架中封装了一套经常操作方式，替换 PHP 的 pcntl，PHP自带的pcntl，存在很多不足。

//

//// php pcntl不足

pcntl没有提供进程间通信的功能。

pcntl不支持重定向标准输入和输出。

pcntl只提供了fork这样原始的接口，容易使用错误。

//

Swoft 基于 Swoole 进程操作封装，功能与 Swoole 完全一样，建议开发者使用 Swoft 的封装操作，方便框架一起迭代升级。2.0.4+ 支持且需要安装 swoft-process( http://swoft.org/docs/2.x/zh-CN/process/index.html ) 组件。

//

(1)、方法列表

所有操作方法，全部在 Swoft\Process\Process 里面，\_\_construct、start、name、exec、...。

--

2、用户进程

( http://swoft.org/docs/2.x/zh-CN/process/user-process.html )

Http/RPC/Websocket/TCP 等服务有些业务场景，需要一个后台运行进程去监控、上报或者其它特殊操作，此时可以在相应服务启动的时候，添加一个用户自定义工作进程，来实现。 自定义用户进程与服务一起启动，服务关闭一起退出，如果自定义用户进程被意外关闭，服务会重新启动一个新的自定义用户进程，保证自定义用户进程一直存在。2.0.4+ 支持且需要安装 swoft-process 组件。

//

(1)、声明用户进程

使用自定义用户进程之前，必须定义用户进程，如下定义一个监控上报信息的用户进程为例。

//

///1/ 自定义用户进程入口

//

<?php declare(strict\_types=1);

namespace App\Process;

use App\Model\Logic\MonitorLogic;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Db\Exception\DbException;

use Swoft\Process\Process;

use Swoft\Process\UserProcess;

/\*\*

 \* Class MonitorProcess

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class MonitorProcess extends UserProcess

{

    /\*\*

     \* @Inject()

     \*

     \* @var MonitorLogic

     \*/

    private $logic;

    /\*\*

     \* @param Process $process

     \*

     \* @throws DbException

     \*/

    public function run(Process $process): void

    {

        $this->logic->monitor($process);

    }

}

?>

自定义用户进程必须实现 Swoft\Process\UserProcess 接口。自定义用户进程必须使用 @Bean 标记为一个 bean 对象。

//

///2/ 业务处理

<?php declare(strict\_types=1);

namespace App\Model\Logic;

use App\Model\Entity\User;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Db\Exception\DbException;

use Swoft\Log\Helper\CLog;

use Swoft\Process\Process;

use Swoft\Redis\Redis;

use Swoole\Coroutine;

/\*\*

 \* Class MonitorProcessLogic

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class MonitorLogic

{

    /\*\*

     \* @param Process $process

     \*

     \* @throws DbException

     \*/

    public function monitor(Process $process): void

    {

        $process->name('swoft-monitor');

        while (true) {

            $connections = context()->getServer()->getSwooleServer()->connections;

            CLog::info('monitor = ' . json\_encode($connections));

            // Database

            $user = User::find(1)->toArray();

            CLog::info('user='.json\_encode($user));

            // Redis

            Redis::set('test', 'ok');

            CLog::info('test='.Redis::get('test'));

            Coroutine::sleep(3);

        }

    }

}

?>

自定义用户进程里面，开发者必须实现类似 `while(true)` 的业务，且里面可以直接使用 Swoft 封装好的所有IO操作，比如数据库、缓存、RPC，以及其它非 Swoft 封装的协程操作。

//

(2)、配置

定义好了用户进程，必须配置才会有效，Http/RPC/Websocket/TCP 服务配置自定义进程都一样，这里以如上自定义的用户进程配置为例。

//

///1/ 配置自定义进程

return [

  'httpServer'     => [

          'class'    => HttpServer::class,

          'port'     => 18306,

          'listener' => [

              'rpc' => bean('rpcServer')

          ],

          'process' => [

              'monitor' => bean(MonitorProcess::class)

          ],

          // ...

      ],

 ];

这里以注入的方式配置了一个自定义用户进程，名称为 monitor。如果配置成功，服务启动后，用户进程里面的业务会自动执行，无需其它操作。

--

3、进程池

( http://swoft.org/docs/2.x/zh-CN/process/process-pool.html )

进程池一般用于需要程序一直运行的场景，比如队列消费，数据计算。Swoft 框架中，基于 Swoole 进程池模型再次封装，便于开发者快速简单的使用进程池。(2.0.4+ 支持且需要安装 swoft-process 组件)

//

(1)、配置

组件安装成功后，默认不需要配置也可以使用，配置如下所述。

//// app/bean.php

return [

    'processPool' => [

        'class' => \Swoft\Process\ProcessPool::class,

        'workerNum' => 3

    ]

];

// 详细参数

workerNum worker 进程数。

ipcType IPC类型。

coroutine 是否开启协程，默认是开启。

Swoft 框架中必须是协程模式运行，协程模式运行下，可以使用 Swoft 封装的所有 IO 操作，以及其它非 Swoft 协程操作。

//

(2)、注解

///1/ @Process

@Process 标记类是一个进程池处理流程。

// 属性列表

workerId int|array 绑定的进程ID,可以是单个或者数组。默认情况，是绑定到其它未绑定的进程。

//

(3)、声明工作进程

配置好之后，就是声明工作进程。如下以 workerNum = 3，定义三个 worker 进程为例。

//

///1/ worker 进程1

<?php declare(strict\_types=1);

namespace App\Process;

use Swoft\Log\Helper\CLog;

use Swoft\Process\Annotation\Mapping\Process;

use Swoft\Process\Contract\ProcessInterface;

use Swoole\Coroutine;

use Swoole\Process\Pool;

/\*\*

 \* Class Worker1Process

 \*

 \* @since 2.0

 \*

 \* @Process(workerId=0)

 \*/

class Worker1Process implements ProcessInterface

{

    /\*\*

     \* @param Pool $pool

     \* @param int  $workerId

     \*/

    public function run(Pool $pool, int $workerId): void

    {

        while (true) {

            CLog::info('worker-' . $workerId);

            Coroutine::sleep(3);

        }

    }

}

?>

worker 进程必须实现 Swoft\Process\Contract\ProcessInterface 接口。

开发者业务必须自己实现类似 while(true) 逻辑。

@Process 注解 workerId=0 表示第1个进程绑定这个处理逻辑流程。

//

///2/ worker 进程2和进程3

<?php declare(strict\_types=1);

namespace App\Process;

use App\Model\Entity\User;

use Swoft\Db\Exception\DbException;

use Swoft\Log\Helper\CLog;

use Swoft\Process\Annotation\Mapping\Process;

use Swoft\Process\Contract\ProcessInterface;

use Swoft\Redis\Redis;

use Swoole\Coroutine;

use Swoole\Process\Pool;

/\*\*

 \* Class Worker2Process

 \*

 \* @since 2.0

 \*

 \* @Process(workerId={1,2})

 \*/

class Worker2Process implements ProcessInterface

{

    /\*\*

     \* @param Pool $pool

     \* @param int  $workerId

     \*

     \* @throws DbException

     \*/

    public function run(Pool $pool, int $workerId): void

    {

        while (true) {

            // Database

            $user = User::find(1)->toArray();

            CLog::info('user='.json\_encode($user));

            // Redis

            Redis::set('test', 'ok');

            CLog::info('test='.Redis::get('test'));

            CLog::info('worker-' . $workerId.' context='.context()->getWorkerId());

            Coroutine::sleep(3);

        }

    }

}

?>

worker 进程必须实现 Swoft\Process\Contract\ProcessInterface 接口。

开发者业务必须自己实现类似 while(true) 逻辑。

@Process 注解 workerId={1,2} 表示第2个进程和第3个进程，同时绑定这个处理逻辑流程。

workerId 绑定ID 是从 0 开始算起, workerId 如果不写默认情况，当前程序流程绑定到其它未绑定的进程。

//

(4)、运行

配置和声明工作进程完成后，就是启动进程池，启动进程池和其它服务启动很类似。

///1/ 非后台启动

php bin/swoft process:start

//

///2/ 后台启动

[root@swoft swoft]# php bin/swoft process:start -d

2019/07/16-09:44:34 [INFO] Swoft\SwoftApplication:setSystemAlias(496) Set alias @base=/data/www/swoft

... ...

2019/07/16-09:44:44 [INFO] Swoft\Processor\ConsoleProcessor:handle(39) Console command route registered (group 13, command 40)

//

///3/ 重启所有 worker 进程

[root@swoft swoft]# php bin/swoft process:reload

2019/07/16-09:45:52 [INFO] Swoft\SwoftApplication:setSystemAlias(496) Set alias @base=/data/www/swoft

... ...

2019/07/16-09:45:59 [INFO] Swoft\Processor\ConsoleProcessor:handle(39) Console command route registered (group 13, command 40)

Server bin/swoft is reloading

Process pool bin/swoft reload success

//

///4/ 重新启动

[root@swoft swoft]# php bin/swoft process:restart

2019/07/16-09:45:28 [INFO] Swoft\SwoftApplication:setSystemAlias(496) Set alias @base=/data/www/swoft

... ...

2019/07/16-09:45:37 [INFO] Swoft\Processor\ConsoleProcessor:handle(39) Console command route registered (group 13, command 40)

Stopping .. Successful!

Process pool restart success !

//

///5/ 停止服务

[root@swoft swoft]# php bin/swoft process:stop

2019/07/16-09:46:35 [INFO] Swoft\SwoftApplication:setSystemAlias(496) Set alias @base=/data/www/swoft

... ...

2019/07/16-09:46:45 [INFO] Swoft\Processor\ConsoleProcessor:handle(39) Console command route registered (group 13, command 40)

Stopping .. Successful!

--

--

八、任务

( http://swoft.org/docs/2.x/zh-CN/task/index.html )

0、任务概述

某些场景对主流程没有依赖，可以直接使用任务来实现类似这些功能。框架为开发者提供了协程和异步两种任务。切记无论是协程任务还是异步任务，任务里面操作都只支持协程，且只能使用框架封装的所有 IO 操作(数据库、缓存...)。

//

(1)、协程任务

协程任务投递任务的时候不会阻塞主进程相当于一次协程调用，一般用于需要等待任务结果返回的场景。

//

(2)、异步任务

有些场景主流程并不关心的任务执行的结果，此时就可以使用异步任务。

--

1、配置与启用

( http://swoft.org/docs/2.x/zh-CN/task/setting.html )

任务配置参数，可以直接在对应的 Server->setting 配置即可，如果要启用任务更简单，Server 新增一个 on 事件。

//

(1)、协程任务

///1/ Http Server 配置开启任务为例

return [

    'httpServer' => [

        // ...

        'on'       => [

            SwooleEvent::TASK   => \bean(TaskListener::class),  // Enable task must task and finish event

            SwooleEvent::FINISH => \bean(FinishListener::class)

        ],

        /\* @see HttpServer::$setting \*/

        'setting'  => [

            'task\_worker\_num'       => 12,

            'task\_enable\_coroutine' => true

        ]

    ],

]

task\_enable\_coroutine 必须为 true。

task 事件和finish 事件必须配置，且为 TaskListener::class 和 FinishListener::class。

//

///2/ Rpc Server 配置开启任务为例

return [

    'rpcServer' => [

        // ...

        'on'       => [

            SwooleEvent::TASK   => \bean(TaskListener::class),  // Enable task must task and finish event

            SwooleEvent::FINISH => \bean(FinishListener::class)

        ],

        /\* @see HttpServer::$setting \*/

        'setting'  => [

            'task\_worker\_num'       => 12,

            'task\_enable\_coroutine' => true

        ]

    ],

]

//

///3/ wsServer Server 配置开启任务为例

return [

    'wsServer' => [

        // ...

        'on'       => [

            SwooleEvent::TASK   => \bean(TaskListener::class),  // Enable task must task and finish event

            SwooleEvent::FINISH => \bean(FinishListener::class)

        ],

        /\* @see HttpServer::$setting \*/

        'setting'  => [

            'task\_worker\_num'       => 12,

            'task\_enable\_coroutine' => true

        ]

    ],

]

//

任务配置与启用，在 Http Server / Rpc Server / Websocket Server 都完全一样，启用任务需要监听 task finish 两个事件。

//

(2)、同步阻塞任务

Swoft 不仅提供协程任务，并且支持同步任务，同步任务和协程任务只能选择一种运行，两种不能同时存在。同步任务只需配置 task 事件，不支持异步 finish 事件。官方建议使用协程任务实现业务， 如果需要通过任务实现MongoDB、PostgreSQL 类似这种场景才使用同步任务。(2.0.4+ 支持)

//

//// 如下以 Http-server 为例

return [

    'httpServer' => [

        // ...

        'on'       => [

            SwooleEvent::TASK   => bean(SyncTaskListener::class),  // Enable sync task

        ],

        /\* @see HttpServer::$setting \*/

        'setting'  => [

            'task\_worker\_num'       => 6,

            'task\_enable\_coroutine' => false

        ]

    ],

]

task\_enable\_coroutine 必须设置为 false，task 事件必须是 SyncTaskListener::class。

同步阻塞任务，不能直接使用框架提供的所有 IO 操作(数据库、缓存、RPC等等)以及应用日志，控制器日志可以使用。 同步阻塞任务的定义和使用与协程任务一样，但是没有上下文。

--

2、声明一个任务

( http://swoft.org/docs/2.x/zh-CN/task/statement.html )

//

///1/ 声明任务：使用任务前，必须定义任务，定义任务很简单，如下定一个任务。

/\*\*

 \* Class TestTask

 \*

 \* @since 2.0

 \*

 \* @Task(name="testTask")

 \*/

class TestTask

{

    /\*\*

     \* @TaskMapping(name="list")

     \*

     \* @param int    $id

     \* @param string $default

     \*

     \* @return array

     \*/

    public function getList(int $id, string $default = 'def'): array

    {

        return [

            'list'    => [1, 3, 3],

            'id'      => $id,

            'default' => $default

        ];

    }

    /\*\*

     \* @TaskMapping()

     \*

     \* @param int $id

     \*

     \* @return bool

     \*/

    public function delete(int $id): bool

    {

        if ($id > 10) {

            return true;

        }

        return false;

    }

}

?>

//

(1)、@Task

标记类是一个任务，name 指定任务名称，默认全路径类名。

//

(2)、@TaskMapping

映射名称，name 名称映射，默认就类的方法名称。

//

//// 注意

|1>、被 @Task 标记类的每个方法就是一个任务，如果方法没有使用 @TaskMapping 注解，不会解析成任务。

|2>、Task投递前，会经过 Swoft\Task\Packet::pack() 方法被json\_encode，因此，投递entity时，task获得的是entity的数组。同时，投递无法被json\_encode的参数会导致报错 （如果是异步任务且没有开启额外的日志，可能效果是task直接结束而没有报错信息）。

--

3、协程任务

( http://swoft.org/docs/2.x/zh-CN/task/coroutine.html )

协程任务投递提供了两种方式，单个投递和批量投递，单个投递是在批量投递的基础之上封装的。如下协程任务投递(备注：co/async方法第三个参数传递为数组，当任务声明的参数为数组时，co/async第三个参数需要 [[xxx,xxx,xxx]]使用)。

//

use Swoft\Task\Task;

// 协程投递

$data = Task::co('testTask', 'list', [12]);

// 协程投递

$result = Task::co('testTask', 'delete', [12]);

//

(1)、任务投递

Task::co(string $name, string $method, array $params = [], float $timeout = 3, array $ext = [])

单个任务投递，返回数据和任务方法返回的数据完全一致类型也一样。

name 投递任务任务名称。

method 投递任务的方法名称。

params 任务传递的参数即是任务方法的参数，数组格式传递。

timeout 超时时间，默认 3s 超时。

ext 任务扩展信息，会传递给任务进程里面。

//

// 多个任务投递格式

$tasks = [

    [

        'taskName',

        'method',

        ['params']

    ]

];

Task::cos(array $tasks, float $timeout = 3, array $ext = [])

tasks 多个任务集合，格式如上。

timeout 超时时间，默认 3s 超时。

ext 任务扩展信息，会传递给任务进程里面。

//

(2)、任务上下文

有些场景需要在任务里面拿到任务的详细信息，这些信息全部在上下文里面。此时可以使用全局函数 context() 获取 Swoft\Task\TaskContext 上下文对象。上下文提供两个方法，分别获取 Swoft\Task\Request 与 Swoft\Task\Response 对象，里面包含投递任务的所有信息。

$request = context()->getRequest();

$response = context()->getRespone();

注意：一定要在任务里面获取上下文，否则获取的是其它环境的上下文。

//

///1/ 任务 Request

namespace Swoft\Task;

class Request implements RequestInterface

{

   // ...

}

//

// 方法列表

getServer 获取任务 Server 信息。

getTaskId 获取任务 ID，对应 Swoole 任务 ID。

getSrcWorkerId 任务来自的 workerId。

getData 投递任务的原始是数据。

getName 任务名称。

getMethod 任务方法。

getParams 任务参数。

getExt 任务扩展信息。

getExtKey 根据 key 快速获取用户信息。

getType 任务类型。

getTaskUniqid 任务全局唯一ID。

--

4、异步任务

( http://swoft.org/docs/2.x/zh-CN/task/async.html )

异步任务一般用于不需要结果的场景且异步去执行，不影响主流程。如下异步任务投递。

//

use Swoft\Task\Task;

$data = Task::async('testTask', 'list', [12]);

$data = Task::async('testTask', 'delete', [12]);

//

(1)、任务投递

Task::async(string $name, string $method, array $params = [], array $ext = [], int $dstWorkerId = -1, callable $fallback = null)

异步任务投递，返回一个全局唯一的任务ID。

name 投递任务任务名称。

method 投递任务的方法名称。

params 任务传递的参数即是任务方法的参数，数组格式传递。

ext 任务扩展信息，会传递给任务进程里面。

dstWorkerId 投递的进程 workerId，默认底层按需选择进程 workerId。

//

(2)、任务上下文

任务上下文和协程任务章节讲解的完全一样。

//

(3)、异步任务结果

有很多情况不需要关注异步任务处理结果，但是也有部分场景需要关注异步任务处理结果，框架为开发者提供了一种事件监听的方式来处理异步任务结果。此事件和普通事件完全一样。如下定义是事件监听。

//

//// 异步任务完成事件监听

<?php

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Log\Helper\CLog;

use Swoft\Task\TaskEvent;

/\*\*

 \* Class FinishListener

 \*

 \* @since 2.0

 \*

 \* @Listener(event=TaskEvent::FINISH)

 \*/

class FinishListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        CLog::info(\context()->getTaskUniqid());

    }

}

?>

事件必须监听 TaskEvent::FINISH 如果需要获取数据可以从上下文中获取，注意此时获取的是任务完成的上下文对象与任务上下对象不一样。

//

(4)、异步任务完成上下文

在异步任务完成监听器里面可以通过 content() 全局函数获取上下文 Swoft\Task\FinishContext 对象。

$taskData = context()->getTaskData();

$taskId = context()->getTaskId();

$taskUniqid = context()->getTaskUniqid();

$server = context()->getServer();

//

getTaskData 任务处理的结果内容。

getTaskId 任务 ID，对应 Swoole 任务 ID。

getTaskUniqid 全局任务唯一ID，框架生成，与任务投递时的全局任务ID一样。

getServer 获取任务 Server 相关信息。

--

5、定时任务

( http://swoft.org/docs/2.x/zh-CN/task/crontab.html )

在某些情况下需要定时的去执行某些任务，通常我们会使用 Linux 系统自带的 Crontab 去定时的执行我们编写好的脚本，但是这样及其不方便，首先 Linux 系统默认的 Crontab 最小单位只能支持到分钟，无法支持秒级任务，其次，如果我们重新编写脚本，则不能很方便友好的复用框架内的资源，如 Mysql 连接资源，框架中的各种类库。针对以上问题，框架为我们内置了一个 Crontab 组件，可以支持秒级任务。( 可用自 >= v2.0.5 )

//

(1)、安装

// 使用定时任务前，必须安装，安装如下所示

composer require swoft/crontab

//

(2)、注解

在 Swoft 中，定时任务的使用非常的简单，只需要使用相关注解定义你的任务类即可。

//

///1/ @Scheduled()

用于声明定时任务，如果是声明定时任务类，则必须使用此注解。

// 参数

name 任务类的名称，为空则为此类的完整 namespace路径。

// 使用示例

@Scheduled()、@Scheduled("taskName")、@Scheduled(name="taskName")

//

///2/ @Cron()

声明需要运行的方法，如果没有使用此注解，则该方法不会被运行。

// 参数

value 任务的 Crontab 表达式，支持到秒。

// 使用示例

@Cron("\* \* \* \* \* \*")、@Cron(value="\* \* \* \* \* \*")，表达式可简写，例如一个每秒都要执行的任务则可定义为 @Cron("\*")。

//

(3)、Cron格式说明

\*    \*    \*    \*    \*    \*

-    -    -    -    -    -

|    |    |    |    |    |

|    |    |    |    |    +----- day of week (0 - 6) (Sunday=0)

|    |    |    |    +----- month (1 - 12)

|    |    |    +------- day of month (1 - 31)

|    |    +--------- hour (0 - 23)

|    +----------- min (0 - 59)

+------------- sec (0-59)

//

//// 示例

* \* \* \* \* \* 表示每秒执行一次。

0 \* \* \* \* \* 表示每分钟的第0秒执行一次，即每分钟执行一次。

0 0 \* \* \* \* 表示每小时的0分0秒执行一次，即每小时执行一次。

0/10 \* \* \* \* \* 表示每分钟的第0秒开始每10秒执行一次。

10-20 \* \* \* \* \* 表示每分钟的第10-20秒执行一次。

10,20,30 \* \* \* \* \* 表示每分钟的第10,20,30秒各执行一次。

//

(4)、声明定时任务

在 Swoft 中使用定时任务相当的简单，只需两步操作，声明定时任务和配置启用，这两部操作都相当的简单，我们先来看声明任务。

<?php declare(strict\_types=1);

namespace App\Crontab;

use Swoft\Crontab\Annotaion\Mapping\Cron;

use Swoft\Crontab\Annotaion\Mapping\Scheduled;

/\*\*

 \* Class CronTask

 \*

 \* @since 2.0

 \*

 \* @Scheduled()

 \*/

class CronTask

{

    /\*\*

     \* @Cron("\* \* \* \* \* \*")

     \*/

    public function secondTask()

    {

        printf("second task run: %s ", date('Y-m-d H:i:s', time()));

    }

    /\*\*

     \* @Cron("0 \* \* \* \* \*")

     \*/

    public function minuteTask()

    {

        printf("minute task run: %s ", date('Y-m-d H:i:s', time()));

    }

}

?>

//

(5)、配置启用

定时任务的执行是基于 Swoft 的 进程,所以我们需要和使用 用户进程 的方式一样在配置中启用 Crontab 组件的自定义进程即可。

return [

  'httpServer'     => [

          // ...

          'process' => [

              'crontab' => bean(Swoft\Crontab\Process\CrontabProcess::class)

          ],

          // ...

      ],

];

如上我们就配置成功了服务启动后，我们的定时任务进程也会随之启动。

//

(6)、手动执行

除了定时执行我们设置好的任务外，我们还可以在业务代码中直接手动执行我们的定时任务，方法如下。

$crontab = BeanFactory::getBean("crontab");

$crontab->execute("testCrontab", "method");

通过 Bean 容器拿到 crontab 管理器，然后直接使用 execute($beanName,$methodName) 方法，此方法有两个参数,$beanName 就是传入在 @Scheduled() 注解中设置的名字，$methodName 则是传入 @Scheduled() 标注的类中，@Cron() 标注的方法。

--

--

九、数据库

--

--

0、数据库前言

( http://swoft.org/docs/2.x/zh-CN/db/index.html )

Swoft 2.x DB操作的方法 高度兼容 Laravel ，让上手更变得简单。

//

(1)、事务

开始事务后，之间的所有操作都在同一个事务里面，但是不支持并发操作，因为是同一个连接。

//

(2)、查询构造器

查询器是一套封装面向对象的方法，来实现SQL拼装和操作。目前仅实现了 MySQL 客户端 采用原生 PDO 方式， 其他类型的数据库客户端可以自己实现，参考配置章节。

//

(3)、实体

实体只有一点小的改动，参考模型章节。

//

(4)、SQL语句

2.x 没有直接可以获取 之前之后的 SQL，可以 调用 toSql() 方法查看。或者在 Swoft\Db\Query\Processor\Processor 查看正在运行的 sql。

--

--

1、配置 (数据库：入门)

( http://swoft.org/docs/2.x/zh-CN/db/setting.html )

//

(1)、简介

Swoft DB 操作高度兼容 Laravel 能使用 原生的 SQL、流畅的查询构造器，和 Eloquent ORM 在从此与 DB 交互变得简单，去掉了复杂的对象关联模型。采用原生 PDO 方式连接数据库。

说下为什么这次要采用PDO 的原生方式？使用mysqlnd模式的pdo、mysqli扩展会加入Hook监听,如果未启用mysqlnd将不支持协程化。也即使说 IO 操作会被自动转换和 swoole 的 MySQL 协程客户端一样。让开发变得简单，更贴近传统框架。

//

(2)、基础配置

数据库的配置放置在 app\bean.php文件中，去掉了繁琐的.env文件配置，你可以认为配置的 db 是一个 bean 对象。

return [

    'db'         => [

       'class'     => Database::class,

       'dsn'       => 'mysql:dbname=test;host=127.0.0.1',

       'username'  => 'root',

       'password'  => '123456',

       'charset'   => 'utf8mb4',

    ],

];

配置方式类似 yii2 对象属性注入的方式的配置，可以通过 \bean('db') 来获取当前配置的Database对象。

class 指定当前 bean容器使用哪一个类 当然你也可以指定自己实现的database类。

dsn PDO需要使用的连接配置信息。

username 数据库登录用户名。

password 数据库登录密码。

charset 数据库字符集。

//

(3)、属性配置详解

//// 详细配置例子

'db'  => [

    'class'    => Database::class,

    'dsn'      => 'mysql:dbname=swoft;host=127.0.0.1',

    'username' => 'root',

    'password' => '123456',

    'charset'  => 'utf8mb4',

    'prefix'   => 't\_',

    'options'  => [

        \PDO::ATTR\_CASE => \PDO::CASE\_NATURAL,

    ],

    'config'   => [

        'collation' => 'utf8mb4\_general\_ci',

        'strict'    => false,

        'timezone'  => '+8:00',

        'modes'     => 'NO\_ENGINE\_SUBSTITUTION,STRICT\_TRANS\_TABLES',

        'fetchMode' => PDO::FETCH\_ASSOC,

    ],

],

//

prefix 表名的公共前缀或后缀。

//

options PDO属性选项。

/\*\*

 \* 默认的PDO连接选项。当然你可以选择替换它

 \*

 \* @var array

 \*/

$options = [

    \PDO::ATTR\_CASE              => \PDO::CASE\_NATURAL,

    \PDO::ATTR\_ERRMODE           => \PDO::ERRMODE\_EXCEPTION,

    \PDO::ATTR\_ORACLE\_NULLS      => \PDO::NULL\_NATURAL,

    \PDO::ATTR\_STRINGIFY\_FETCHES => false,

    \PDO::ATTR\_EMULATE\_PREPARES  => false,

    \PDO::ATTR\_ERRMODE           => \PDO::ERRMODE\_EXCEPTION,

];

//

config MySQL的其他配置， 默认驱动仅实现了 MySQL 如果需要其他数据库驱动请参考Connector & Connection( http://swoft.org/docs/2.x/zh-CN/db/setting.html#Connector&Connection )。

collation 设置指定数据集如何排序。

timezone 设置时区设置。

modes 设置连接模式（可以是一维数组，也可以是英文逗号分割的 modes）。

strict 设置获取查询以启用严格模式（实际上也是设置modes配置）。

fetchMode 设置 pdo 返回的类型 默认的连接池返回的类型为 FETCH\_ASSOC 也就是数组方式, 其他类型参考官方手册( https://php.net/manual/en/pdo.constants.php )。

//

默认的db.pool连接池的DB类查询查询操作，返回的是数组。自定义的db连接池如果也需要返回数组，需要设置fetchMode参数 为 PDO::FETCH\_ASSOC，否则返回是的 stdClass 对象。

//

(4)、读写&连接

'db2'  => [

    'class'  => Database::class,

    'charset'  => 'utf8mb4',

    'prefix'   => 't\_',

    'options'  => [],

    'config'   => [

       'collation' => 'utf8mb4\_general\_ci',

       'strict'    => false,

       'timezone'  => '+8:00',

       'modes'     => 'NO\_ENGINE\_SUBSTITUTION,STRICT\_TRANS\_TABLES',

    ],

    'writes' => [

        [

            'dsn'      => 'mysql:dbname=swoft;host=127.0.0.1',

            'username' => 'root',

            'password' => '123456',

        ],

    ],

    'reads'  => [

        [

            'dsn'      => 'mysql:dbname=swoft;host=127.0.0.1',

            'username' => 'root',

            'password' => '123456',

        ],

    ],

],

//

writes 主库配置 执行Create Update Delete 操作的时候会从连接池 随机选择主库节点执行。

reads 从库配置 Read 操作使用默认会从连接池 随机选择一个从库节点执行 如果想强制查询的主库，那么请在链式操作中使用 useWritePdo() 方法即可。

//

如果你想公用配置请参考基础配置( http://swoft.org/docs/2.x/zh-CN/db/setting.html#%E5%9F%BA%E7%A1%80%E9%85%8D%E7%BD%AE )。 默认会覆盖合并dsn' , 'username' ,'password','charset' ,'prefix'  ,'options', 'config这些选项配置公用，所以在writes/reads中配置变化的值就可以了，你也可以覆盖上层配置。

//

(5)、连接池配置

想必用的 1.0 的小伙伴对连接池并不陌生，2.x 对它进行了简化配置，连接池的好处也不用多说，更好的管理资源、对数据库的保护。

db 的连接是通过 连接池创建和释放的，通过ConnectionManager类来管理连接，创建的为短链接 操作执行失败会重试一次。

每当调用 toSql() 方法或者执行完操作的时候都会归还连接到连接池中，默认是连接池名称是 db.pool 使用的 database 是\bean('db') 也就是基础配置配置的，连接池配置放置在 app\bean.php文件中。

每一个 `worker` 都会创建一个同样的连接池。并不是越多越好，参数配置要根据，机器配置和`worker` 个数衡量。

//

//// 下面我们看看连接池，如何自定义一个连接池？

'db.pool2' => [

    'class'       => \Swoft\Db\Pool::class,

    'database'    => \bean('db2'),

    'minActive'   => 10,

    'maxActive'   => 20,

    'maxWait'     => 0,

    'maxWaitTime' => 0,

    'maxIdleTime' => 60,

]

//

class 是默认的 Pool 对象 你可以更具官方的自己继承实现，然后换成自己的 Pool 类名就可以了。

database 驱动的数据库对象，是读 & 写连接配置的。

minActive 连接池需要维持的连接数。

maxActive 连接池最大保持的连接数。

maxWait 连接池最多等待连接数, 如果没有限制为0(默认)。

maxWaitTime 连接最大等待时间，单位秒，如果没有限制为0(默认)。

maxIdleTime 连接最大空闲时间，单位秒。

//

(6)、Connector&Connection

连接器和连接的关系是创建连接必备的。

//

///1/ Connector

Connector 主要是用于，根据配置创建真实的 PDO 创建连接。

//

///2/ Connection

Connection主要用于，数据库的语法解析，设置表前缀，获取默认查询语法实例，重连错误判断。

Swoft 默认仅提供 MySQL 的Connector&Connection。为什么呢？因为swoole暂且将 pdo\_pgsql，pdo\_ori，pdo\_odbc，pdo\_firebird 这些 PDO扩展加入 底层 Hook。也就是说使用 pdo\_pgsql，pdo\_ori，pdo\_odbc，pdo\_firebird 执行的 IO 操作不会让出 CPU 资源是同步执行的，执行期间协程不会上下文切换。

如果想要使用 pgsql（你可以用 Swoole 的协程 PgSQL 客户端）也是可以的 只需实现Connector和Connection，参照 MySQL 的实现方式即可。

//

(7)、关于分库分表的一些见解

Swoft DB 暂未提供分库和分表 的方案，后续会提供dbSelect interface 实现接口来选择不同的database， 目前暂时可以通过配置db.pool 连接池和db 连接来选择不同数据库。切换连接池详见：选择连接池( https://www.swoft.org/docs/2.x/zh-CN/db/builder.html#%E9%80%89%E6%8B%A9%E8%BF%9E%E6%8E%A5%E6%B1%A0 )。

//

(8)、关于去掉DB服务化的见解

Swoft 2.x去掉了 服务发现相关的东西，融入框架显得笨重，后续会提供 dbProvide interface 来提供获取连接地址接口。

--

--

2、模型

( http://swoft.org/docs/2.x/zh-CN/db/model.html )

(1)、简介

无论是高级查询还是基础查询，都会需要一个表实体。一个表字段和一个类属性是一一映射，对类的操作相当于对表的操作，该类称为一个实体，Swoft 2.x 比 1.x 实体使用简单得许多，兼有Builder查询构造器所有的方法，使用和查询构造器的方法一致，只是返回可能变成了一个 实体。

//

(2)、实体

一个实体类对应一张数据库的表结构，一个实体对象代表了表的一行数据记录。注意： 实体不能作为属性被注入到任何类, 因为每个实体对象都是不同的数据记录行。实体对象都是在哪用就在哪里创建它。

//

///1/ 实体定义

<?php declare(strict\_types=1);

 namespace SwoftTest\Db\Testing\Entity;

 use Swoft\Db\Annotation\Mapping\Column;

 use Swoft\Db\Annotation\Mapping\Entity;

 use Swoft\Db\Annotation\Mapping\Id;

 use Swoft\Db\Eloquent\Model;

 /\*\*

  \* Class User

  \*

  \* @since 2.0

  \*

  \* @Entity(table="user",pool="db.pool2")

  \*/

 class User extends Model

 {

     /\*\*

      \* @Id(incrementing=true)

      \*

      \* @Column(name="id", prop="id")

      \* @var int|null

      \*/

     private $id;

     /\*\*

      \* @Column(name="password", hidden=true)

      \* @var string|null

      \*/

     private $pwd;

     /\*\*

      \* @Column()

      \*

      \* @var int|null

      \*/

     private $age;

     /\*\*

      \* @Column(name="user\_desc", prop="udesc")

      \*

      \* @var string|null

      \*/

     private $userDesc;

    /\*\*

     \* @return string|null

     \*/

     public function getUserDesc(): ?string

     {

        return $this->userDesc;

     }

    /\*\*

     \* @param string|null $userDesc

     \*/

     public function setUserDesc(?string $userDesc): void

     {

        $this->userDesc = $userDesc;

     }

     /\*\*

      \* @return int|null

      \*/

     public function getId(): ?int

     {

         return $this->id;

     }

     /\*\*

      \* @param int|null $id

      \*/

     public function setId(?int $id): void

     {

         $this->id = $id;

     }

     /\*\*

      \* @return int|null

      \*/

     public function getAge(): ?int

     {

         return $this->age;

     }

     /\*\*

      \* @param int|null $age

      \*/

     public function setAge(?int $age): void

     {

         $this->age = $age;

     }

     /\*\*

      \* @return string|null

      \*/

     public function getPwd(): ?string

     {

         return $this->pwd;

     }

     /\*\*

      \* @param string|null $pwd

      \*/

     public function setPwd(?string $pwd): void

     {

         $this->pwd = $pwd;

     }

}

也可以使用 devtool 里面的快速生成实体工具。如果没有定义 @Column 的列, 使用插入/更新 的不存在@Column值, 将会被框架自动过滤。

//

(3)、注解标签

///1/ @Entity

标记一个类是一个实体，有两个参数：name 定义该实体映射的数据库表名（必填）；pool 该实体选择的 连接池默认为 db.pool 你可以替换成自己定义的连接池，设计这个参数的目的有 2 个，第一个你可以切换自己定义db 连接池，第二是你可以使用自己实现的数据库驱动。假如 User 表 是 MySQL 的， Count 表 可以是 PostSQL 的使用不同的连接池即可实现。

//

///2/ @Column

标记一个列，如果一个列没有定义@Column那么查询它将不会显示，这样即使你新增了数据库字段也不会影响生产环境运行。

name 定义类属性映射的表字段，没该注解标记的属性，不映射(默认为字段名为属性名)。

prop 为字段设置一个别名(prop 只是为字段设置一个别名，只有在调用toArray的时候才会被转换。这样能隐藏数据库真实的字段。使用where等子句，需要使用数据库字段)。

hidden 是否隐藏，如果为真那么它 toArray() 的时候将会被隐藏，但是不影响你通过 Getter获取它，你也可以调用实体的addVisible方法将他取消隐藏。

说明：所有字段属性，必须要有getter和setter方法，你可以使用phpstorm 快捷键 ctrl+n，它会更具属性 快速生成 getter和setter。

注意：若表字段有下划线，类属性均定义为 小驼峰 写法 例： 字段 user\_name 则属性写为 $userName。

2.x 去掉了 type 属性 现在会使用 属性上定义的 @var 注解定义的第一个类型，决定了返回值类型，底层会强转类型。

//

///3/ @Id

该注解标明当前类属性对应了数据库表中的主键，必须有这个注解标记，不能设置多个@Id注解。incrementing 是否为递增主键，默认为递增主键。

//

(4)、Prop 操作 (2.0.6 支持)

模型插入支持，使用 prop 插入。例如 上面例子中实体, 真实数据库字段是 user\_desc, prop 字段是 udesc, 底层会自动转化成 user\_desc 插入。当然这不影响之前的使用。

User::new([

    'udesc' => $desc,

])->save();

条件使用 prop，使用 whereProp 方法, whereProp 方法和可以用 where 一样使用。

$where      = [

    'pwd' => md5(uniqid()),

    ['udesc', 'like', 'swoft%'],

    ['whereIn', 'id', [1]]

];

// 'select \* from `user` where (`password` = ? and `user\_desc` like ? and `id` in (?))';

$sql = User::whereProp($where)->toSql();

where 扩展使用, 数据里面每一个元素, 为方法名, 支持 Query Builder 里面的所有与 Where 相关的方法。

$toSql = 'select \* from `user` where (`id` in (?) or `id` = ? or `status` > ? and `age` between ? and ?)';

$where = [

    ['whereIn', 'id', [1]],

    ['orWhere', 'id', 2],

    ['orWhere', 'status', '>', -1],

    ['whereBetween', 'age', [18, 25]]

];

$sql   = User::where($where)->toSql();

// same as

User::where('id', '=', [1])

            ->orWhere('id', 2)

            ->orWhere('status', '>', -1)

            ->whereBetween('age', [18, 25])

//

(5)、插入数据

///1/ 对象方式插入获取自增Id

$user = User::new();

$user->setName('name');

$user->setSex(1);

$user->setDesc('this my desc');

$user->setAge(mt\_rand(1, 100));

$user->save();

// saved after getId()

$id = $user->getId();

//

///2/ 数组方式

$attributes = [

    'name'      => uniqid(),

    'password'  => md5(uniqid()),

    'age'       => mt\_rand(1, 100),

    'user\_desc' => 'u desc'

];

$user  = User::new($attributes)

$result3 =$user->save();

$id = $user->getId()

在新增save 之后可以用 getter 方法获取自增 id.

//

///3/ 批量插入

如果你想批量插入可以使用 User::insert([])方法，使用和查询构造器的 insert方法 使用完全一致。

//

(6)、删除数据

///1/ 指定 id 删除

$user = User::find($id);

$result = $user->delete();

//

///2/ 使用条件删除

$result = User::where('id', 1)->delete();

//

///3/ 删除一条

$result = User::where('stauts',1 )->limit(1)->delete();

//

(7)、实体更新

//// 使用 setter 或者array 都可以更新

$user = User::find($id);

$name   = uniqid();

$user->setAge(1);

$result = $user->update(['name' => $name]);

//

(8)、条件批量更新

//// 更新一条数据

$wheres   = [

    'name' => 'swoft',

    ['id', '>=', 2]

];

$orWheres = [

    ['status', '<>', '1']

];

$result   = User::where($wheres)

                ->limit(1)

                ->orWhere($orWheres)

                ->update(['status' => 1]);

//

(9)、更新/插入

//// 可以使用updateOrCreate 返回的是一个实体

$user = User::updateOrCreate(['id' => 1], ['age' => 18, 'name' => 'uname']);

echo $user->getName();

//

//// 可以用 updateOrInsert 返回一个 bool 值

$isOk = User::updateOrInsert(['id' => 1], ['age' => 18, 'name' => 'uname']);

//

(10)、使用主键进行批量更新

//// 在这例子中 id 是 User实体的 @Id() 主键

$values = [

    ['id' => 1, 'age' => 18],

    ['id' => 2, 'age' => 19],

];

User::batchUpdateByIds($values);

使用批量更新 必须指定主键的值, 框架会根据主键的值进行 批量更新。

//

(11)、快速更新

如果已知道 更新的 主键 id 可以使用 modifyById 方法进行快速更新。

// method 1

$row = User::modifyById($id, ['user\_desc' => $expectLabel]);

// method 2

User::find($id)->update(['user\_desc' => $expectLabel]);

//

如果不知道需要更新的主键 id 可以使用modify 方法,该方法会先根据条件查找 id 再进行 update。

$where  = ['user\_desc' => 'CP'];

$values = ['user\_desc' => $expectLabel];

// method 1

$row = User::modify($where, $values);

// method 2

$model = User::where($where)->first()->update($values);

方法 1 和方法 2 是相同的意思。

//

(12)、递增/递减

///1/ 单个字段 递增/递减

使用 increment 递增，decrement 递减, 第三个参数是附带更新的值。

$updateByWhereId = User::where('id', 1)->increment('age', 1);

$updateByModel   = User::find(1)->decrement('age', 2);

//

///2/ 多个字段 递增/递减

使用 updateAllCounters 方法更新. 需要注意的是, 请谨慎使用更新条件,最好使用主键更新以免造成表锁.

User::updateAllCounters(['user\_desc' => $expectLabel], ['age' => -1]);

//

///3/ 如果知道 需要更新的 主键id 可以使用 updateAllCountersById 方法

// method 1

User::updateAllCountersById((array)$id, ['age' => 1], ['user\_desc' => $expectLabel]);

// method 2

User::find($id)->updateCounters(['age' => -1]);

//

(13)、查询数据

模型的查询方法和查询构造器完全兼容。

//// 查询一条数据，返回一个实体

// 方法 1

$user =  User::find(1, ['id','name']);

// 方法 2

$user = User::where('id',1)->first();

//

//// 查询多条数据

copy

// 方法 1

$users = User::findMany([1,2,3,4], ['id','name']);

// 方法 2

$useer = User::whereIn('id', [1,2,3,4])->select(['id','name'])->get();

//

如果你想获取的列表中都是 实体对象 可以使用 getModels 方法，该方法返回的是一个实体数组 你可以这样使用。

$users = User::where('id', 22)->getModels(['id', 'age']);

/\* @var User $user \*/

foreach ($users as $user) {

    $age = $user->getAge();

}

//

有时候我们需要按数据库某个键作为 key 我们逻辑映射关系，可以使用 Collection 的 keyBy 方法。

//// 假如 需要第一页 数据需要以id作为key 你可以这样使用。

$users = User::forPage(1, 10)->get(['id', 'age'])->keyBy('id');

/\* @var User $user \*/

foreach ($users as $id => $user) {

    $age = $user->getAge();

}

//

//// 实体使用 Join 系列操作，不会返回实体

$userCounts = User::join('count', 'user.id', '=', 'count.user\_id')->get();

//

(14)、分块结果

如果你需要处理数千个 Eloquent 记录，可以使用 chunk 命令。chunk 方法会检索 Eloquent 模型的「分块」，将它们提供给指定的 Closure 进行处理。在处理大型结果集时，使用 chunk 方法可节省内存。

 Flight::chunk(200, function ($flights) {

        foreach ($flights as $flight) {

            //

        }

 });

 传递到方法的第一个参数是希望每个「分块」接收的数据量。闭包则被作为第二个参数传递，它会在每次执行数据库查询传递每个块时被调用。

//

(15)、使用游标

cursor 允许你使用游标来遍历数据库数据，该游标只执行一个查询。处理大量数据时，可以使用 cursor 方法可以大幅度减少内存的使用量。

foreach (Flight::where('foo', 'bar')->cursor() as $flight) {

    //

}

//

(16)、找不到」异常

如果你希望在找不到模型时抛出异常，可以使用 findOrFail 以及 firstOrFail 方法。这些方法会检索查询的第一个结果。如果没有找到相应结果，就会抛出一个 DbException。

$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();

//

(17)、赋值

如果你觉得 setter 太麻烦了可以使用,批量填充功能, 使用这种方式要注意如果该字段没有匹配到 @Column 值将会被忽 这样能保证安全的更新和插入。

// Properties

$attributes = [

    'name'      => uniqid(),

    'password'  => md5(uniqid()),

    'age'       => mt\_rand(1, 100),

    'user\_desc' => 'u desc'

];

// one

$result3 = User::new($attributes)->save();

// two

$result3 = User::new()->fill($attributes)->save();

//

(18)、检索集合

你还可以使用 查询构造器 提供的 count、sum、max 以及其它 聚合函数。这些方法只会返回适当的标量值而不是整个模型实例。

$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');

//

(19)、其他创建方法

///1/ firstOrCreate/ firstOrNew

你还可以使用其他两种方法来创建模型：firstOrCreate 和 firstOrNew。

firstOrCreate 方法会使用给定的字段及其值在数据库中查找记录。如果在数据库中找不到模型，则将使用第一个参数中的属性以及可选的第二个参数中的属性插入记录。

irstOrNew 方法就类似 firstOrCreate 方法，会在数据库中查找匹配给定属性的记录。如果模型未被找到，则会返回一个新的模型实例。请注意，在这里面，firstOrnew 返回的模型还尚未保存到数据库，必须要手动调用 save 方法才能保存它。

// 通过 name 属性检索航班，当结果不存在时创建它...

$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// 通过 name 属性检索航班，当结果不存在的时候用 name 属性和 delayed 属性去创建它

$flight = App\Flight::firstOrCreate(

    ['name' => 'Flight 10'], ['delayed' => 1]

);

// 通过 name 属性检索航班，当结果不存在时实例化...

$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// 通过 name 属性检索航班，当结果不存在的时候用 name 属性和 delayed 属性实例化

$flight = App\Flight::firstOrNew(

    ['name' => 'Flight 10'], ['delayed' => 1]

);

//

///2/ updateOrCreate

你也可能会遇到想要更新现有模型或创建新模型（如果不存在）的情况。Swoft 提供了 updateOrCreate 方法来完成该操作，像 firstOrCreate 方法一样，updateOrCreate 方法会保存模型，所以不需要调用 save()。

// 如果有从奥克兰飞往圣地亚哥的航班，将价格设为 99 美元，如果不存在匹配的模型就创建一个。

$flight = App\Flight::updateOrCreate(

    ['departure' => 'Oakland', 'destination' => 'San Diego'],

    ['price' => 99]

);

更多方法请参照 查询构造器( http://swoft.org/docs/2.x/zh-CN/db/builder.html )。

//

(20)、自动写入时间戳

默认情况下，Eloquent 会默认数据表中存在 created\_at 和 updated\_at 这两个字段。如果你不需要自动更新这两个字段，则需要在模型内将 $modelTimestamps 属性设置为 false。

<?php

namespace App;

use App\Model\Entity;

class User

{

    /\*\*

     \* Whether the model is automatically maintained with a timestamp

     \*

     \* @var bool

     \*/

    public $modelTimestamps = false;

}

?>

//

如果你需要自定义时间戳格式，可在模型内设置 $modelDateFormat 属性。这个属性决定了日期属性应如何存储在数据库中，以及模型被序列化成数组或 JSON 时的格式。

<?php

    class User

    {

        /\*\*

         \* The storage format of the model's date field

         \*

         \* @var string

         \*/

        protected $modelDateFormat = 'Y-m-d H:i:s';

    }

?>

//

如果需要自定义用于存储时间戳的字段名，可在模型中通过设置 CREATED\_AT 和 UPDATED\_AT 常量来实现。

时间戳 支持 数据库 int 和 timestamp 类型, 底层会自动根据实体的属性 CREATED\_AT 和 UPDATED\_AT 这个两个字段定义的 @var 来判断. 用户无需操心生成时间戳格式.

<?php

    class User

    {

       protected const CREATED\_AT = 'create\_time';

       protected const UPDATED\_AT = 'update\_data';

    }

?>

暂不支持注解继承 Dao 层继承 Entity 暂不可用。如果要让框架自动维护 CREATED\_AT 和 UPDATED\_AT , 那么这两个常量对应的字段, 必须要有与之对应的 getter 和 setter。

//

(21)、事件

Eloquent 的模型触发了几个事件，可以在模型的生命周期的以下几点进行监控： creating、created、updating、updated、saving、saved、deleting、deleted。事件能在每次在数据库中保存或更新特定模型类时轻松地执行代码。当然你完全可以通过 AOP 来实现它。

当新模型第一次被保存时， creating 以及 created 事件会被触发。如果模型已经存在于数据库中并且调用了 save 方法，会触发 updating 和 updated 事件。在这两种情况下，saving / saved 事件都会触发。

//

//// 事件名称是 swoft.model+模型名+动作名

模型名 是首字母默认会小写 例如实体名称 SendMessage 要监听它的saving 动作的话 格式就是 swoft.model.sendMessage.saving 其他模型也类似.

//

可以监听某个模型的saving操作的动作, 也可以监听所有模型的saving动作

//// 监听模型单个动作

<?php declare(strict\_types=1);

namespace App\Listener;

use App\Model\Entity\User;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

/\*\*

 \* Class UserSavingListener

 \*

 \* @since 2.0

 \*

 \* @Listener("swoft.model.user.saving")

 \*/

class UserSavingListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        /\* @var User $user \*/

        $user = $event->getTarget();

        if ($user->getAge() > 100) {

            // stopping saving

            $event->stopPropagation(true);

            $user->setAdd(100);

        }

    }

}

?>

//

//// 监听所有模型的单个动作

<?php declare(strict\_types=1);

namespace App\Listener;

use App\Model\Entity\User;

use Swoft\Db\DbEvent;

use Swoft\Db\Eloquent\Model;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

/\*\*

 \* Class RanListener

 \*

 \* @since 2.0

 \*

 \* @Listener(DbEvent::MODEL\_SAVED)

 \*/

class ModelSavedListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        /\* @var Model $modelStatic \*/

        $modelStatic = $event->getTarget();

        if ($modelStatic instanceof User) {

            // to do something....

        }

        // ....

    }

}

?>

//

// 公共的事件名列表, 可以在Swoft\Db\DbEvent 类中参看所有事件

( Event Params Description )

swoft.db.transaction.begin  没有参数  事务启动。

swoft.db.transaction.commit 没有参数  事务提交。

swoft.db.transaction.rollback 没有参数  事务回滚。

swoft.model.saving  target 是具体操作实体类 所有实体保存中事件。

swoft.model.saved target 是具体操作实体类 所有实体保存后事件。

swoft.model.updating  target 是具体操作实体类 所有实体更新前事件。

swoft.model.updated target 是具体操作实体类 所有实体更新后事件。

swoft.model.creating  target 是具体操作实体类 所有实体创建前事件。

swoft.model.created target 是具体操作实体类 所有实体创建后事件。

swoft.model.deleting  target 是具体操作实体类 所有实体删除前事件。

swoft.model.deleted target 是具体操作实体类 所有实体后删除前事件。

swoft.db.ran  target 是连接对象,参数 1=未预处理 sql ,参数 2=绑定的参数  所有 sql 执行后的事件,事件返回的连接已返回给连接池只能获取它的配置信息。

swoft.db.affectingStatementing  target 是连接对象,参数 1=正在处理的PDO statement ,参数 2=绑定的参数  正在执行 update 和delete动作

swoft.db.selecting  target 是连接对象,参数 1=正在处理的PDO statement ,参数 2=绑定的参数  正在执行查询动作。

如果是正在进行时(ing) 在监听事件中是调用了 $event->stopPropagation(true); 后续操作会终止直接返回结果. 对过去式停止无效。

//

(22)、FQA

使用模型 使用 select 方法最好, 不要使用 as 不然查询结果与实体映射可能会有问题。使用模型的方法 更新/插入的值 都会过滤处理 没有 定义 @Column 的值将会被过滤。

--

--

3、查询器 (查询构造器)

( http://swoft.org/docs/2.x/zh-CN/db/builder.html )

//

(1)、简介

Swoft 的数据库查询构造器为创建和运行数据库查询提供了一个方便的接口。它可用于执行应用程序中大部分数据库操作，且可在所有支持的数据库系统上运行。

Swoft 的查询构造器使用 PDO 参数绑定来保护您的应用程序免受 SQL 注入攻击。因此没有必要清理作为绑定传递的字符串。

你可以使用 DB::table('xxxx')得到一个 Builder 对象 也可以使用 Builder::new()->from('xxx') 这两种写法返回结果是一样的，Builder对象不会分配连接，只有执行 sql 的时候才会从连接池从获取。

//

(2)、获取结果

///1/ 从一个数据表中获取所有行

你可以 DB 上使用 table 方法来开始查询。该 table 方法为给定的表返回一个查询构造器实例，允许你在查询上链式调用更多的约束，最后使用 get 方法获取结果。

// sql = select \* from `user`

$users = DB::table('user')->get();

//

该 get 方法返回一个包含 Collection 的结果，其中每个结果都是 Array。你可以访问字段作为对象的属性来访问每列的值。

foreach ($users as $user) {

    echo $user->name;

}

你还可以使用 Collection 所有方法 Collection 提供了强大的对象操作方法。

//

如果想更快的查询全部的数据可以用cursor方法，底层采用 yield 实现。其中每个结果都是 Array。

$users = DB::table('user')->cursor();

foreach ($users as $user){

  echo $user->name;

}

//

///2/ 从数据表中获取单行或单列

/// first()方法

如果你只需要从数据表中获取一行数据，你可以使用 first 方法，该方法返回一个 Array。

$user = DB::table('user')->where('name', 'Sakura')->first();

if ($user) {

    echo $user->name;

}

//

/// value()方法

如果你甚至不需要整行数据，则可以使用 value 方法从记录中获取单个值。该方法将直接返回该字段的值。

$name = DB::table('users')->where('name', 'Sakura')->value('name');

//

///3/ 获取一列的值

/// pluck()方法

如果你想获取包含单列值的集合，则可以使用 pluck 方法。在下面的例子中，我们将获取角色表中标题的集合。

$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {

    echo $title;

}

你还可以在返回的集合中指定字段的自定义键值。

$roles = DB::table('users')->pluck('email', 'name');

foreach ($roles as $name => $email) {

    echo $email;

}

//

(3)、分块结果

如果你需要处理上千条数据库记录，你可以考虑使用 chunk 方法。该方法一次获取结果集的一小块，并将其传递给 闭包 函数进行处理。在修复数据的时候就很适用。例如，我们可以将全部 user 表数据切割成一次处理 100 条记录的一小块。

DB::table('users')->orderBy('id')->chunk(100, function (\Swoft\Stdlib\Collection $users) {

    foreach ($users as $user) {

       echo $user->name;

    }

});

你可以通过在 闭包 中返回 false 来终止继续获取分块结果。

DB::table('users')->orderBy('id')->chunk(100, function (\Swoft\Stdlib\Collection $users) {

    // Process the records...

    return false;

});

闭包里面传递的 $users 是一个Collection对象，each方法也是通过 chunk 实现的 只是参数不同位置相反。

//

(4)、聚合

查询构造器还提供了各种聚合方法，比如 count, max，min， avg，还有 sum。你可以在构造查询后调用任何方法。

$userNum = DB::table('users')->count();

$price   = DB::table('orders')->max('price');

当然，你也可以将这些聚合方法与其他的查询语句相结合。

$price = DB::table('orders')

                ->where('status', 1)

                ->avg('price');

如果没有查询到任何数据 返回值是一个 int 类型的 0。avg是average方法的别名，而且返回是一个 float|int 类型。

count 固定返回 int, max|min|avg|sum 这些函数可能涉及浮点数计算，底层没有强制转换，返回类型为float|int，这个值是数据库返回的值。

//

//// 判断记录是否存在 exists()方法、doesntExist()方法

除了通过 count 方法可以确定查询条件的结果是否存在之外，还可以使用 exists 和 doesntExist 方法。

 return DB::table('orders')->where('id', 1)->exists();

 return DB::table('orders')->where('id', 1)->doesntExist();

//

(5)、查询

///1/ select()方法：指定一个 Select 语句；

当然你可能并不总是希望从数据库表中获取所有列。使用 select 方法，你可以自定义一个 select 查询语句来查询指定的字段。

// select `name`, `age` as `user\_age` from `user`

$users = DB::table('user')->select('name', 'age as user\_age')->get();

//

///2/ distinct()方法：distinct 方法会强制让查询返回的结果不重复；

$users = DB::table('users')->distinct()->get();

//

///3/ addSelect()方法：如果你已经有了一个查询构造器实例，并且希望在现有的查询语句中加入一个字段，那么你可以使用 addSelect 方法；

$query = DB::table('users')->select('name');

$users = $query->addSelect(['age'])->get();

//

(6)、原生表达式

///1/ selectRaw()方法：有时候你可能需要在查询中使用原生表达式。你可以使用 selectRaw方法 创建一个原生表达式。

// select count(\*) as `user\_count`, `name` from `user`

$users = DB::table('user')

  ->selectRaw('count(\*) as `user\_count`, `name`'));

  ->get();

提示 原生表达式将会被当做字符串注入到查询中，因此你应该小心使用，避免创建 SQL 注入的漏洞。

//

///2/ whereRaw / orWhereRaw

whereRaw 和 orWhereRaw 方法将原生的 where 注入到你的查询中。这两个方法的第二个参数还是可选项，值还是绑定参数的数组。

// select `name`, `age` as `user\_age` from `user` where age > 18

$users = DB::table('user')

    ->whereRaw('age > :age', ['age' => 18])

    ->select('name', 'age as user\_age')

    ->get();

//

///3/ havingRaw / orHavingRaw

havingRaw 和 orHavingRaw 方法可以用于将原生字符串设置为 having 语句的值。

$orders = DB::table('user')

  ->selectRaw('sum(age) as age')

  ->groupBy('user\_desc')

  ->havingRaw('age > ?', [17])

  ->get();

//

///4/ orderByRaw

orderByRaw 方法可用于将原生字符串设置为 ,order by 子句的值。

$time = time();

$orderBy = 'if(`dead\_time`>' . $time . ', update\_time,0) desc,create\_time desc';

$orders = DB::table('ticket')

    ->orderByRaw($orderBy)

    ->get();

//

///5/ fromRaw

fromRaw 方法的自定义 FROM 关键字参数,比如使用强制索引。

$sql = DB::table('')

  ->select('id', 'name')

  ->fromRaw('`user` force index(`idx\_user`)')

  ->get();

//

(7)、Joins

///1/ Inner Join Clause

查询构造器也可以编写 join 方法。若要执行基本的 「内链接」，你可以在查询构造器实例上使用 join 方法。传递给 join 方法的第一个参数是你需要连接的表的名称，而其他参数则使用指定连接的字段约束。

// 你还可以在单个查询中连接多个数据表

$users = DB::table('users')

  ->join('contacts', 'users.id', '=', 'contacts.user\_id')

  ->join('orders', 'users.id', '=', 'orders.user\_id')

  ->select('users.\*', 'contacts.phone', 'orders.price')

  ->get();

//

///2/ Left Join 语句

如果你想使用 「左连接」代替「内连接」 ，可以使用 leftJoin 方法。leftJoin 方法与 join 方法用法相同。

$users = DB::table('users')

  ->leftJoin('posts', 'users.id', '=', 'posts.user\_id')

  ->get();

//

///3/ Cross Join 语句

使用 crossJoin 方法和你想要连接的表名做 「交叉连接」。交叉连接在第一个表和被连接的表之间会生成笛卡尔积。

// select \* from `user` cross join `count` on `count`.`user\_id` = `user`.`id`

$users =Builder::new()

  ->from('user')

  ->crossJoin('count', 'count.user\_id', '=', 'user.id')

  ->get();

//

///4/ 高级 Join 语句

你可以指定更高级的 join 语句。比如传递一个 闭包 作为 join 方法的第二个参数。此 闭包 接收一个 JoinClause 对象，从而指定 join 语句中指定的约束。

DB::table('users')

  ->join('contacts', function ($join) {

      $join->on('users.id', '=', 'contacts.user\_id')->orOn(...);

  })

  ->get();

//

如果你想要在连接上使用「where」 风格的语句，你可以在连接上使用 where 和 orWhere 方法。这些方法会将列和值进行比较，而不是列和列进行比较。

DB::table('users')

  ->join('contacts', function ($join) {

      $join->on('users.id', '=', 'contacts.user\_id')

           ->where('contacts.user\_id', '>', 5);

  })

  ->get();

//

///5/ 子连接查询

你可以使用 joinSub，leftJoinSub 和 rightJoinSub 方法关联一个查询作为子查询。他们每一种方法都会接收三个参数：子查询，表别名和定义关联字段的闭包。

$latestPosts = DB::table('posts')

                   ->select('MAX(created\_at) as last\_created\_at')

                   ->where('is\_published', true)

                   ->groupBy('user\_id');

// $latestPosts 是一个 query 对象

$users = DB::table('users')

        ->joinSub($latestPosts, 'latest\_posts', function($join) {

            $join->on('users.id', '=', 'latest\_posts.user\_id');

        })->get();

//

///6/ Unions

查询构造器还提供了将两个查询 「联合」 的快捷方式。比如，你可以先创建一个查询，然后使用 union 方法将其和第二个查询进行联合。

// (select \* from `user`) union all (select \* from `user`) union (select \* from `user`)

Builder::new()

    ->from('user')

    ->unionAll(function (Builder $builder) {

        $builder->from('user');

    })

    ->union(Builder::new()->from('user'))

    ->get();

提示: 你也可以使用 unionAll 方法，用法和 union 方法是的一样。

//

(8)、简单的 Where 语句

在构造 where 查询实例中，你可以使用 where 方法。调用 where 最基本的方式是需要传递三个参数：第一个参数是列名，第二个参数是任意一个数据库系统支持的运算符，第三个是该列要比较的值。

///1/ 验证 「money」 字段的值等于 100 的查询

$users = DB::table('user')->where('money', '=', 100)->get();

//

///2/ 为了方便，如果你只是简单比较列值和给定数值是否相等，可以将数值直接作为 where 方法的第二个参数。

$users = DB::table('users')->where('votes', 100)->get();

//

///3/ 可以使用其他的运算符来编写 where 子句。

$users = DB::table('users')

                ->where('votes', '>=', 100)

                ->get();

$users = DB::table('users')

                ->where('votes', '<>', 100)

                ->get();

$users = DB::table('users')

                ->where('name', 'like', 'T%')

                ->get();

//

///4/ 可以传递条件数组到 where 函数中。

$users = DB::table('users')->where([

    ['status', '=', '1'],

    ['subscribed', '<>', '1'],

])->get();

//

///5/ 混合数组 where 形式，数组里面再嵌套一个数组也是可以的。

$wheres   = [

    'name' => 'cyb',

    ['status', '>=', 2],

    ['money', '>', 0, 'or']

];

// select \* from `user` where (`name` = ? and `status` >= ? or `money` > ?)

$users    = User::where($wheres)->get();

//

(9)、Or 语句

你可以一起链式调用 where 约束，也可以在查询中添加 or 字句。 orWhere 方法和 where 方法接收的参数一样。

$users = DB::table('user')

  ->where('money', '>', 100, 'or')

  ->orWhere('name', 'John')

  ->get();

//

(10)、其他 Where 语句

///1/ whereBetween

// whereBetween 方法验证字段值是否在给定的两个值之间

$users = DB::table('user')->whereBetween('money', [1, 100])->get();

//

///2/ whereNotBetween

// whereNotBetween 方法验证字段值是否在给定的两个值之外。

$users = DB::table('user')

  ->whereNotBetween('money', [1, 100])

  ->get();

//

///3/ whereIn / whereNotIn

// whereIn 方法验证字段的值必须存在指定的数组里

$users = DB::table('user')

  ->whereIn('id', [1, 2, 3])

  ->get();

//

// whereNotIn 方法验证字段的值必须不存在于指定的数组里

$users = DB::table('user')

    ->whereNotIn('id', [1, 2, 3])

    ->get();

//

///4/ whereNull / whereNotNull

// whereNull 方法验证指定的字段必须是 NULL

$users = DB::table('user')

    ->whereNull('created')

    ->get();

//

//// whereNotNull 方法验证指定的字段必须不是 NULL

$users = DB::table('users')

    ->whereNotNull('created')

    ->get();

//

///5/ whereDate / whereMonth / whereDay / whereYear / whereTime

// 使用这些日期函数要注意 MySQL 时区， whereDate 方法用于比较字段值与给定的日期。

$users = DB::table('users')

  ->whereDate('created', '2018-09-08')

  ->get();

//

// whereMonth 方法用于比较字段值与一年中指定的月份

$users = DB::table('users')

  ->whereMonth('created', '9')

  ->get();

//

// whereDay 方法用于比较字段值与一月中指定的日期

$users = DB::table('users')

  ->whereDay('created\_at', '20')

  ->get();

//

// whereYear 方法用于比较字段值与指定的年份

$users = DB::table('users')

  ->whereYear('created\_at', '2018')

  ->get();

//

// whereTime 方法用于比较字段值与指定的时间（时分秒）

$users = DB::table('users')

  ->whereTime('created\_at', '=', '11:20:45')

  ->get();

//

///6/ whereColumn

// whereColumn 方法用于比较两个字段的值 是否相等

$users = DB::table('users')

  ->whereColumn('first\_name', 'last\_name')

  ->get();

//

// 可以传入一个比较运算符

$users = DB::table('users')

  ->whereColumn('updated\_at', '>', 'created\_at')

  ->get();

//

// whereColumn 你也可以传递数组 用 and 运算符链接

$users = DB::table('users')

  ->whereColumn([

      ['first\_name', '=', 'last\_name'],

      ['updated\_at', '>', 'created\_at']

  ])->get();

//

(11)、参数分组

有时候你需要创建更高级的 where 子句，例如「where exists」或者嵌套的参数分组。 Swoft 的查询构造器也能够处理这些。下面，让我们看一个在括号中进行分组约束的例子。

DB::table('user')

  ->where('name', '=', 'John')

  ->where(function ($query) {

    $query->where('votes', '>', 100)

          ->orWhere('title', '=', 'Admin');

  })

  ->get();

你可以看到，通过一个 Closure 写入where 方法构建一个查询构造器 来约束一个分组。这个 Closure 接收一个查询实例，你可以使用这个实例来设置应该包含的约束。上面的例子将生成以下 SQL。

select \* from `user` where `name` = 'sakura' and (`money` > 100 or `title` = 'test')

//

(12)、WhereExists

whereExists 方法允许你使用 where exists SQL 语句。 whereExists 方法接收一个 Closure 参数，该 whereExists 方法接受一个 Closure 参数，该闭包获取一个查询构建器实例从而允许你定义放置在 exists 字句中查询。

DB::table('users')

  ->whereExists(function ($query) {

      $query->from('orders')

            ->whereRaw('orders.user\_id = users.id');

  })

  ->get();

上述查询将产生如下的 SQL 语句。

select \* from `users`

where exists (

    select \* from `orders` where `orders.user\_id` = `users.id`

)

//

(13)、JsonWhere

Swoft 也支持查询 JSON 类型的字段（仅在对 JSON 类型支持的数据库上）。目前，本特性仅支持 MySQL 5.7+。

// 使用 -> 操作符查询 JSON 数据

$users = DB::table('users')

                ->where('options->language', 'en')

                ->get();

$users = DB::table('users')

                ->where('preferences->dining->meal', 'cookie')

                ->get();

//

// 你也可以使用 whereJsonContains 来查询 JSON 数组

$users = DB::table('users')

  ->whereJsonContains('options->languages', 'en')

  ->get();

//

// MySQL 的 whereJsonContains 可以支持多个值

$users = DB::table('users')

  ->whereJsonContains('options->languages', ['en', 'de'])

  ->get();

//

(14)、Ordering,Grouping,Limit,Offset

///1/ orderBy

orderBy 方法允许你通过给定字段对结果集进行排序。 orderBy 的第一个参数应该是你希望排序的字段，第二个参数控制排序的方向，可以是 asc 或 desc：

//

///2/ latest / oldest

latest 和 oldest 方法可以使你轻松地通过日期排序。它默认使用 created\_at 列作为排序依据。当然，你也可以传递自定义的列名。

$user = DB::table('users')

  ->oldest()

  ->first();

//

///3/ inRandomOrder

inRandomOrder 方法被用来将结果随机排序。例如，你可以使用此方法随机获得一条记录。

$randomUser = DB::table('user')

  ->inRandomOrder()

  ->first();

//

///4/ groupBy / having

groupBy 和 having 方法可以将结果分组。 having 方法的使用与 where 方法十分相似。

$users = DB::table('users')

  ->selectRaw("count(\*) count")

  ->groupBy('type')

  ->having('count', '>', "100")

  ->get();

//

// 你可以向 groupBy 方法传递多个参数。

$users = DB::table('users')

  ->groupBy('first\_name', 'status')

  ->having('account\_id', '>', "100")

  ->get();

对于更高级的 having 语法，参见 havingRaw 方法。

//

///5/ skip / take

要限制结果的返回数量，或跳过指定数量的结果，你可以使用 skip 和 take 方法。

$users = DB::table('users')->skip(10)->take(5)->get();

//

//// 或者你也可以使用 limit 和 offset 方法

$users = DB::table('users')

  ->offset(10)

  ->limit(5)

  ->get();

//

// forPage()方法：如果想快速的分页查询 可以使用 forPage 方法；

// $this->skip(($page - 1) \* $size)->take($size); = forPage($page, $size)

DB::table('users')

  ->forPage($page, $size)

  ->get();

//

(15)、条件语句

有时候你可能想要子句只适用于某个情况为真时才执行查询。例如你可能只想给定值在请求中存在的情况下才应用 where 语句。你可以通过使用 when 方法。

$role = true;

$users = DB::table('users')

  ->when($role, function ($query) use ($role) {

    return $query->where('role\_id', $role);

  })

  ->get();

//

when 方法只有在第一个参数为 true 的时候才执行给定的闭包。如果第一个参数为 false ，那么这个闭包将不会被执行。你可以传递另一个闭包作为 when 方法的第三个参数。 该闭包会在第一个参数为 false 的情况下执行。为了说明如何使用这个特性，我们来配置一个查询的默认排序。

$sortBy = null;

$users = DB::table('users')

  ->when($sortBy, function ($query) use ($sortBy) {

      return $query->orderBy($sortBy);

  }, function ($query) {

      return $query->orderBy('name');

  })

  ->get();

//

(16)、插入

查询构造器还提供了 insert 方法用于插入记录到数据库中。 insert 方法接收数组形式的字段名和字段值进行插入操作。

DB::table('users')->insert(

    ['email' => 'john@example.com', 'votes' => 0]

);

//

// 甚至可以将数组传递给 insert 方法，将多个记录插入到表中

DB::table('users')->insert([

    ['email' => 'taylor@example.com', 'votes' => 0],

    ['email' => 'dayle@example.com', 'votes' => 0]

]);

//// 自增ID

如果数据表有自增 ID ，使用 insertGetId 方法来插入记录并返回 ID 值。

$id = DB::table('user')->insertGetId([

    'age'  => 18,

    'name' => 'Sakura',

]);

//

(17)、更新

当然， 除了插入记录到数据库中，查询构造器也可以通过 update 方法更新已有的记录。 update 方法和 insert 方法一样，接受包含要更新的字段及值的数组。你可以通过 where 子句对 update 查询进行约束。

//

///1/ 更新 JSON 字段

更新 JSON 字段时，你可以使用 -> 语法访问 JSON 对象中相应的值，此操作只能用于支持 JSON 字段类型的数据库。

DB::table('users')

  ->where('id', 1)

  ->update(['options->enabled' => true]);

//

///2/ 自增与自减

查询构造器还为给定字段的递增或递减提供了方便的方法。此方法提供了一个比手动编写 update 语句更具表达力且更精练的接口。这两种方法都至少接收一个参数：需要修改的列。第二个参数是可选的，用于控制列递增或递减的量。

DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);

//

// 可以在操作过程中指定要更新的字段

DB::table('users')->where('id', 1)->increment('votes', 1, ['updated' => 1]);

//

///3/ 如果你想自定义更新 可以这样

$res = DB::table('user')->where('id', $id)->update([

    'posts' => DB::raw('`posts` + 1'),

    'views' => Expression::new('`views` + 1'),

    'name'  => 'updated',

]);

DB::raw(xxx) 等同 Expression::new(xxx) 使用这两个方法的时候要预防SQL注入。

//

(18)、删除

查询构造器也可以使用 delete 方法从表中删除记录。 在使用 delete 前，可以添加 where 子句来约束 delete 语法。

DB::table('users')->where('votes', '>', 100)->delete();

//

// 如果你需要清空表，你可以使用 truncate 方法，它将删除所有行，并重置自增 ID 为零

DB::table('users')->truncate();

//

(19)、打印最后执行的sql

// 可使用toSql()方法打印最后执行的sql

DB::table('users')->where('id', $id)->toSql();

//

(20)、锁

查询构造器也包含一些可以帮助你在 select 语法上实现「悲观锁定」的函数。若想在查询中实现一个「共享锁」， 你可以使用 读锁 sharedLock 方法。 共享锁可防止选中的数据列被篡改，直到事务被提交为止。

// 共享锁

DB::table('users')->where('votes', '>', 100)->sharedLock()->get();

//

// 或者，你可以使用 写锁 lockForUpdate 方法。使用 「update」锁可避免行被其它共享锁修改或选取。

DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();

//

(21)、选择连接池

如果有多个连接连接池 ，默认分配的连接是在 db.pool默认连接池 里面获取的，如果要获取自己连接池的连接。

// 在 bean 里面配置的连接池名

$poolName = 'pool.order2';

$user = DB::query($poolName)->from('user')->where('id', $id)->get();

DB::query($poolName)方法获得同样是一个 Builder 对象。

//

(22)、连接何时释放

底层只有在 执行 sql 的时候才会从 DB 连接池中拿连接执行，执行之后会自动释放。Builder 对象不在依赖 Connection。释放连接: 把连接还给连接池。

//

(23)、FQA

// 在 where 的闭包中使用这样的例子是错误的

$res = DB::table('user')

   ->where(function (Builder $query) {

       $query->forPage(1, 10)

         ->orderBy('age', 'asc')

         ->where('id', 1);

   })

   ->orderBy('id', 'desc')

   ->get();

在例子闭包中的orderBy和forPage不会生效。只有 where 相关的约束才会生效。

--

4、事务

( http://swoft.org/docs/2.x/zh-CN/db/transaction.html )

(1)、开启事务

开启事务后，事务之间的所有操作都是同一个连接，注意不能使用并发操作。如果你想要开始一个事务，并且对回滚和提交能够完全控制，那么你可以使用 DB 的 beginTransaction 方法。

DB::beginTransaction();

或者

DB::connection()->beginTransaction();

一旦开启了事务，当前连接会绑定到当前的协程环境中，保证提交和回滚，查询都是同一个连接保证数据的安全性，只有提交或者回滚完毕才会解除绑定。不同连接池的事务, 是相互独立的 . 没有存在关系。

//

(2)、事务回滚

如果操作失败需要回滚 使用下面两种方式都可以。

DB::rollBack();

或者

DB::connection()->rollBack();

//

(3)、事务提交

如果操作执行成功需要 使用下面两种方式都可以。

DB::commit();

或者

DB::connection()->commit();

//

(4)、常见问题

///1/ 事务是否支持嵌套

框架的答案是可以做事务嵌套，因为部分数据库支持 支持 savepoints 能力。 在MySQL中, 保存点savepoints属于事务控制处理部分。利用savepoints可以回滚指定部分事务，从而使事务处理更加灵活和精细。如果你进行了事务嵌套嵌套的事务会保存在savepoints里面，可以做嵌套事务的精细控制。

DB::beginTransaction();

$user = User::find($id);

$user->update(['name' => $id]);

DB::beginTransaction();

User::find($id)->update(['name'=>'sakuraovq']);

DB::rollBack();

DB::commit();

嵌套里面的的事务进行了回滚，不会影响外层改变的数据只回滚name=sakuraovq的修改，这段代码执行下来 name = $id

//

///2/ 如果事务没有提交怎么办

DB::connection()->beginTransaction();

$user = User::find($id);

\sgo(function ()use($id) {

    DB::connection()->beginTransaction();

    User::find($id);

});

类似这样的代码如果我们忘记 提交事务/回滚。Swoft 在SwoftEvent::COROUTINE\_DEFER事件中会检查是否还处于事务状态，如果是会自动 rollback到最初开启事务的状态。连接会归还到连接池中，不会造成资源泄露。

//

(5)、错误示范

DB::beginTransaction();

$user = User::find($id);

\sgo(function () use ($id) {

    $user1 = User::find($id);

});

$user->update(['name' => 'sakuraovq'.mt\_rand(110,10000)]);

DB::commit();

类似这样的代码虽然执行没有问题，这种写法是错误的，会造成数据的错乱。请不要在事务中嵌套协程，在执行 db 操作, 上文讲到事务是绑定到当前协程的，切换了协程也就是另一个新的连接。

DB::connection()->beginTransaction();

$user = User::find($id);

$user->update(['name' => 2]);

\sgo(function () use ($user) {

    $user->update(['name' => 1]);

});

DB::rollBack();

这样也是错的，这段代码执行下来你会发现，子协程的修改操作回滚不受控制。因为它们使用了不同的连接，并非绑定在一起。

--

5、原生操作

( http://swoft.org/docs/2.x/zh-CN/db/origin.html )

(0)、运行原生的SQL查询

一旦配置好数据库连接后，便可以使用 DB 对象运行查询。 DB 为每种类型的查询提供了方法： select，selectOne，update，insert，delete , cursor, statement, affectingStatement 和 unprepared。

//

(1)、运行查询语句

$users = DB::select('select \* from `user` where `status` = ?', [1]);

select 方法将始终返回一个数组，数组中的每个结果都是一个 = Array，可以像下面这样访问结果值。

foreach ($users as $user) {

    echo $user->name;

}

//

// 如果只是查询一条可用 selectOne 结果返回是一个 = Array， 如果没有查询到数据返回是一个空的Array。

$sql= 'select \* from `user` where `id` = ?';

$res = DB::selectOne($sql, [1]);

if ($res) {

    echo $res->name;

}

//

//// 如果你最快的方式遍历数据表所有数据 可用使用 DB::cursor($sql) 方法。

$sql= 'select \* from `user`';

$res = DB::cursor($sql);

foreach($res as $user){

   echo $user->name;

}

为什么说是最快的呢，因为底层采用 yield 机制获取数据 比 chunk 快很多。

//

//// 使用命名绑定

除了使用 ? 表示参数绑定外，你也可以使用命名绑定来执行一个查询。

$results = DB::select('select \* from `user` where `id` = :id', ['id' => 1]);

//

(2)、运行插入语句

可以使用 DB Facade 的 insert 方法来执行 insert 语句。与 select 一样，该方法将原生 SQL 查询作为其第一个参数，并将绑定数据作为第二个参数。

DB::insert('insert into users (`id`, `name`) values (?, ?)', [1, 'sumi']);

//

(3)、运行更新语句

update 方法用于更新数据库中现有的记录。该方法返回受该语句影响的行数。

$affected = DB::update('update `user` set `status` = 2 where `name` = ?', ['ovo']);

//

(4)、运行删除语句

delete 方法用于从数据库中删除记录。与 update 一样，返回受该语句影响的行数。

copy

$deleted = DB::delete('delete from `users`');

//

(5)、运行普通语句

有些数据库语句不会有任何返回值。对于这些语句，你可以使用 DB 的 statement 方法来运行。

DB::statement('drop table `user`');

有些数据库语句如果你想知道执行是否有成功。对于这些语句，你可以使用 DB 的 unprepared 方法来运行。

$unprepared = DB::unprepared('DROP TRIGGER IF EXISTS `sync\_to\_item\_table`');

if ($unprepared) {

    // Todo something...

}

只有使用 unprepared 方法才可以执行多条sql, 需要预处理的 不能执行多条 sql。

--

6、切库 (切换数据库)

在之前的2.0.2，之前在一个连接池中没有提供切库的功能，导致不同数据库需要配置多个连接池，这大大增加了维护成本，所以在 2.0.2 提供了切换数据库功能，你可以在链式操作中手动自定 db()方法，这显然不够灵活难以维护。下面推荐一种根据上下文切库的操作(Available: >= v2.0.2)。

( http://swoft.org/docs/2.x/zh-CN/db/selectDb.html )

(1)、DbSelector

<?php declare(strict\_types=1);

namespace App\Common;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Db\Connection\Connection;

use Swoft\Db\Contract\DbSelectorInterface;

/\*\*

 \* Class DbSelector

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class DbSelector implements DbSelectorInterface

{

    /\*\*

     \* @param Connection $connection

     \*/

    public function select(Connection $connection): void

    {

        // 在请求中获取 id

        $selectIndex  = (int)context()->getRequest()->query('id', 0);

        $createDbName = $connection->getDb();

        if ($selectIndex == 0) {

            $selectIndex = '';

        }

        // 数据库名 + id  类似这样的 order\_database\_1, 好处是会根据上下文自动切库

        $dbName = sprintf('%s%s', $createDbName, (string)$selectIndex);

        $connection->db($dbName);

    }

}

?>

//

(2)、DbSelector 配置

使用 DbSelector 必须要实现 DbSelectorInterface 接口。实现了 DbSelector 还没完 还需在bean.php文件配置中定义 dbSelector属性。

'db2'        => [

    'class'      => Database::class,

    'dsn'        => 'mysql:dbname=test;host=127.0.0.1',

    'username'   => 'root',

    'password'   => 'swoft123456',

    'dbSelector' => bean(DbSelector::class)

],

'db2.pool'   => [

    'class'    => Pool::class,

    'database' => bean('db2')

],

dbSelector 是上面例子实现的类, 这样每次操作都会调用它 自动切库, 让分库变得简单。使用这种好处是, 你可以像没有分库一样使用, 大大降低了维护成本. 官方推荐使用这种方案。如果不满足这种的话你只能在操作的时候使用 db() 方法手动指定数据库了, 这样容错成本会大大降低。

// 实体使用方式

User::db('test2')->insertGetId([

          'name'      => uniqid(),

          'password'  => md5(uniqid()),

          'age'       => mt\_rand(1, 100),

          'user\_desc' => 'u desc',

          'foo'       => 'bar',

          'xxxx'      => '223asdf'

]);

// DB 使用方式

DB::table('user')->db('test2')->insertGetId([

          'name'      => uniqid(),

          'password'  => md5(uniqid()),

          'age'       => mt\_rand(1, 100),

          'user\_desc' => 'u desc',

]);

--

--

十、Redis

( http://swoft.org/docs/2.x/zh-CN/redis/index.html )

Redis 是一个开源的，高级键值对存储数据库。由于它包含 字符串 , 哈希 , 列表 , 集合 , 和 有序集合 这些数据类型，所以它通常被称为数据结构服务器。

Swoft Redis 默认驱动是 phpRedis 驱动方式，默认的需要具备 phpRedis 扩展，你可以通过 PECL 安装 PhpRedis PHP 扩展。这个扩展安装起来比较复杂，但对于大量使用 Redis 的应用程序来说可能会产生更好的性能。

如果你需要使用 predis/predis 你也可以的，参考redis 配置章节。

//

1、配置 (Redis 设置)

( http://swoft.org/docs/2.x/zh-CN/redis/setting.html )

//

(1)、基础配置

Swoft 应用的 Redis 配置都在配置文件 app/bean.php 中。在这个文件里，你可以看到 redis 数组里面包含了应用程序使用的 Redis 服务器。

默认的redis.pool连接池是开启自带序列化的。你可以直接 set array，但是 通过 set执行的 key 自增操作将失效。如果你想把 redis 作为一个数据库，而不是缓存你可以采用配置一个新的连接池(自定义的连接池是没有开启序列化)。

'redis'      => [

    'class'    => RedisDb::class,

    'driver'   => 'phpredis',

    'host'     => '127.0.0.1',

    'port'     => 6379,

    'database' => 0,

    // 'password' => 123456,

],

//

默认的服务器配置应该足以进行开发。当然，你也可以根据使用的环境来随意更改这个数组。 只需在配置文件中给每个 Redis 服务器指定名称、host 和 port 即可。

class 指定当前配置驱动类型。

使用哪一个 redis 驱动 默认是 phpredis (php-ext)，你也可以换成 predis 具体看 实现predis( http://swoft.org/docs/2.x/zh-CN/redis/setting.html#%E5%AE%9E%E7%8E%B0predis )。

host 连接地址 默认 127.0.0.1。

port 端口 默认 6379。

database 缓存数据库index 默认 0。

//

(2)、详细配置

'redis'      => [

    'class'         => RedisDb::class,

    'host'          => '127.0.0.1',

    'port'          => 6379,

    'database'      => 0,

    'retryInterval' => 10,

    'readTimeout'   => 0,

    'timeout'       => 2,

    'option'        => [

        'prefix'      => 'xxx',

        'serializer' => Redis::SERIALIZER\_PHP

    ],

],

retryInterval 重试间隔。

readTimeout 读取超时时间 0：不超时，单位秒。

timeout 超时时间 0：不超时，单位秒。

option phpredis 的其他配置，prefix redis 前缀；serializer 序列化支持三种选项(0 为0就会关闭序列化；Redis::SERIALIZER\_NONE 不序列化数据；Redis::SERIALIZER\_PHP 使用php内置序列化/反序列化；Redis::SERIALIZER\_IGBINARY 使用igBinary序列化/反序列化)。

tips：关闭序列化，把 option 选项的 serializer 设置为 0 即可关闭序列化。

//

(3)、Redis集群配置

集群配置与基础配置都是在同一文件下，集群配置增加了 clusters ，连接池会默认使用集群配置创建连接。连接机制会优先使用 集群的配置，如果没有 集群配置，才用的普通的配置。

'redis' => [

    'class'    => \Swoft\Redis\RedisDb::class,

    'option'   => [

        'timeout'    => 3,

        'persistent' => true,

    ],

    'clusters' => [

        [

            'host'         => '127.0.0.1',

            'prefix'       => 'swoft-t',

            'read\_timeout' => 1,

            'port'         => 6379,

            'database'     => 1,

            'password'     => 123456,

        ],

    ]

]

// option 集群配置

timeout 超时时间 默认 0 单位秒

persistent 是否持久化 默认 false

//

// clusters redis clusters 节点二维数组方法配置

host 节点地址。

port 节点端口。

password 节点密码。

read\_timeout 读取超时时间。

database 节点的缓存数据库index。

提示：集群没有序列化配置；

//

(4)、Redis连接池配置

Swoft 所有连接池配置都差不多，如果你想使用不同的 缓存数据库index或者不同节点。 那么就很适合定义连接池，配置都在 app\bean.php里面。

注意：每一个 worker 都会创建一个同样的连接池。并不是越多越好，参数配置要根据，机器配置和 和worker 个数衡量。

//

///1/ 如果是集群的话可以这样做

 'redis-clusters' => [

    'class'    => \Swoft\Redis\RedisDb::class,

    'option'   => [

        'timeout'    => 3,

        'persistent' => true,

    ],

    'clusters' => [

        [

            'host'         => '127.0.0.1',

            'prefix'       => 'swoft-t',

            'read\_timeout' => 1,

            'port'         => 6379,

            'database'     => 1,

            'password'     => 123445,

        ],

    ]

],

'redis.clusters-pool'     => [

    'class'   => \Swoft\Redis\Pool::class,

    'redisDb' => \bean('redis-clusters'),

    'minActive'   => 10,

    'maxActive'   => 20,

    'maxWait'     => 0,

    'maxWaitTime' => 0,

    'maxIdleTime' => 40,

],

//

///2/ 普通 redis

'redis-2'      => [

    'class'         => RedisDb::class,

    'host'          => '127.0.0.2',

    'port'          => 6378,

    'database'      => 1,

    'retryInterval' => 10,

    'readTimeout'   => 0,

    'timeout'       => 2,

    'option'        => [

        'prefix'      => 'xxx',

        'serializer' => Redis::SERIALIZER\_PHP

    ],

],

'redis.pool.2'     => [

    'class'   => \Swoft\Redis\Pool::class,

    'redisDb' => \bean('redis-2'),

    'minActive'   => 10,

    'maxActive'   => 20,

    'maxWait'     => 0,

    'maxWaitTime' => 0,

    'maxIdleTime' => 60,

],

//

/// redis.clusters-pool 连接池名称自定义

class 连接池驱动类 一般不用改，如果需要可以继承写成自己的 Pool 类也是可以的。

redisDb 指定当前连接使用哪一个 redisDb配置， 例子中替换自己定义的。

minActive 连接池需要维持的连接数。

maxActive 连接池最大保持的连接数。

maxWait 连接池最多等待连接数, 如果没有限制为0(默认)。

maxWaitTime 连接最大等待时间，单位秒，如果没有限制为0(默认)。

maxIdleTime 连接最大空闲时间，单位秒。

//

/// 使用 redis.clusters-pool 连接池

Redis::connection('redis.clusters-pool')->get($key);

//

/// 你可以将默认的连接池替换成自己的

'redis.pool'     => [

    'class'   => \Swoft\Redis\Pool::class,

    'redisDb' => \bean('redis-2')

]

//

/// redis.pool 是默认的连接池名称，把连接池里面的 redisDb 属性替换成自己定义的使用就和正常一样了。

Redis::set($key, ["name" => "sakura"]);

//

(5)、实现predis

如果使用predis 需要实现两个契约 Swoft\Redis\Contract\ConnectorInterface和Swoft\Redis\Contract\ConnectionInterface。参考 PhpRedisConnection 和 PhpRedisConnector 的 实现。然后在配置中的 redis 中的'driver' => 'phpredis' 替换成 predis 即可。

提示：推荐使用 phpredis 扩展，因为它更高效率。

//

///1/ 实现步骤

复制 PhpRedisConnection 和 PhpRedisConnector 改名成 PredisConnection 和 PredisConnector，在 vendor/swoft/redis/src/RedisDb.php 中添加如下代码。

/\*\*

 \* @return array

 \*/

public function defaultConnectors(): array

{

    return [

        self::PHP\_REDIS => bean(PhpRedisConnector::class),

        self::P\_REDIS => bean(PredisConnector::class) //^2\_3^实现predis驱动;

    ];

}

/\*\*

 \* @return array

 \*/

public function defaultConnections(): array

{

    return [

        self::PHP\_REDIS => bean(PhpRedisConnection::class),

        self::P\_REDIS => bean(PredisConnection::class) //^2\_3^实现predis驱动;

    ];

}

最后在配置中的 redis 中的'driver' => 'phpredis' 替换成 predis 即可。

--

2、如何使用

( http://swoft.org/docs/2.x/zh-CN/redis/usage.html )

//

(1)、简单操作

///1/ 比如显示给定用户的配置文件。

$user = Redis::get('user:profile:'.$id);

Redis::set('user:profile:'.$id, "ok");

//

你可以在 Redis 上调用任何的 Redis 命令。Swoft 使用魔术方法将传递命令给 Redis服务器，因此只需传递 Redis 命令所需的参数即可。

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);

//

所有方法和 操作 phpredis 原生一样 本质是通过魔术方法代理访问的，让操作变得更简单。当然普通的字符串设置不能满足我们的日常开发，你也可以使用数组的方式。

$setData = [

  'goods' => [

    'goods\_id'   => 1,

    'goods\_name' => 'iPhone xx'

  ]

];

Redis::set($key, $setData);

通过 get 方法 调用，底层会自动反序列化，保证了存进去，取出来的数据一致性。

//

$scores1 = [

    'key1' => 11,

    'key3' => 11,

    'key4' => 11,

    'key2' => 21,

];

// zAdd 11 key1 11 key2 11 key3 11 key4

$result1 = Redis::zAdd('key', $scores1);

key 成员，value 是分数。成员不可以重复，分数可以。

//

(2)、通过 @Inject注入连接池方式使用Redis

///1/ 通过Inject注入属性使用和1.0方式一样

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Redis\Pool;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

/\*\*

 \*

 \* @Controller("Redis")

 \*/

class RedisController

{

    /\*\*

     \* 例子 1: 如果 Inject 没有参数,会使用 var 定义的类型

     \*

     \* @Inject()

     \*

     \* @var Pool 默认连接使用的是 redis.pool

     \*/

    private $redis;

    /\*\*

     \* 例子 2: 如果 Inject 指定参数,会使用指定的 pool 注入到该属性. 和 var 定义的类型没关系

     \*

     \* @Inject("redis.inc.pool")

     \*

     \* @var Pool

     \*/

    private $redisInc;

    /\*\*

     \* @RequestMapping(route="find")

     \*

     \* @return array

     \*

     \* @throws Throwable

     \*/

    public function find(): array

    {

        $this->redis->set('user', ["name"=>"gimi", "age"=>"18"]);

        $this->redisInc->incr('user-count',1);

        return $this->redis->get('user');

    }

}

?>

你可以在 Inject 中指定使用的哪一个redis连接池，如果在Inject中没有指定连接池，将默认会用 @var 指定的类型注入。

//

(3)、指定连接池获取连接

///1/ 当然 所有的连接都是在连接池中分配的，如果你想使用自己定义的连接可以使用

$poolName  = 'redis-clusters-pool'

$redis     =  Redis::connection($poolName);

$redis->get("a");

//

Redis::connection(); 如果没有指定连接池名字，默认会从系统定义的redis.pool连接池中获取连接，该方法 返回的是一个连接，而不是连接池。不要共享连接，要共享连接池。

获取的连接了之后，操作和phpredis 原生使用方式扩展一致。如何创建连接池 参考 redis 设置章节 ，默认是在 redis.pool连接池中获取的。

//

(4)、缓存命中率

如果你想看一个 key 获取的命中情况，可以在你的notice日志级别中获取，格式大致是  redis.hit/req.... 这样的前缀，有助于你分析缓存命中率。

//

(5)、事件监听

///1/ before 事件

// 在Redis 执行之前底层会抛出 RedisEvent::BEFORE\_COMMAND 事件

<?php declare(strict\_types=1);

namespace App\Listener;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Redis\RedisEvent;

/\*\*

 \* Class FinishListener

 \*

 \* @since 2.0

 \*

 \* @Listener(event=RedisEvent::BEFORE\_COMMAND)

 \*/

class RedisBeforeListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        [$method, $parameters] = $event->getParams();

    }

}

?>

//

///2/ after 事件

在 Redis 命令之后会抛出 RedisEvent::AFTER\_COMMAND 事件，你可以监听命令的执行情况，如果在执行命令中出现异常则不会抛出 RedisEvent::AFTER\_COMMAND 事件。

<?php declare(strict\_types=1);

namespace App\Listener;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Redis\RedisEvent;

/\*\*

 \* Class FinishListener

 \*

 \* @since 2.0

 \*

 \* @Listener(event=RedisEvent::AFTER\_COMMAND)

 \*/

class RedisAfterListener implements EventHandlerInterface

{

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        [$method, $parameters, $result] = $event->getParams();

    }

}

?>

--

3、事务 (Redis 事务操作)

( http://swoft.org/docs/2.x/zh-CN/redis/transaction.html )

redis 事务不能回滚，但是能保证原子性，用 lua 脚本也能实现 redis 事务效果。

//

//// 下面是一个事务操作例子，返回数据有点特别，偶数为成功与否，奇数为执行 key，下面写了一个结果遍历方法。

$count  = 2;

$result = Redis::transaction(function (\Redis $redis) use ($count) {

    for ($i = 0; $i < $count; $i++) {

        $key = "key:$i";

        $redis->set($key, $i);

        $redis->get($key);

    }

});

/\*

$result = array(4) {

    [0]=>

    bool(true)

    [1]=>

    int(0)

    [2]=>

    bool(true)

    [3]=>

    int(1)

}\*/

foreach ($result as $index => $value) {

    if ($index % 2 == 0) {

        // is exec ok ？

       if ($value) {

            // todo...

       }

    }

}

--

4、通道 (redis 管道操作)

( http://swoft.org/docs/2.x/zh-CN/redis/pipeline.html )

如果你需要在一个操作中向服务器发送很多命令，推荐你使用管道命令。 pipeline 方法接收一个带有 Redis 实例的 闭包 。 你可以将所有的命令发送给这个 Redis 实例，它们都会一次过执行完。

//

//// 比如连续设置 10 个key，返回的是可以数组，你可以遍历判断是否全部成功。

public function testPipeline()

{

    $count  = 10;

    $result = Redis::pipeline(function (\Redis $redis) use ($count) {

        for ($i = 0; $i < $count; $i++) {

            $redis->set("key:$i", $i);

        }

    });

  // \count($result) ==  $count;

    foreach ($result as $index => $value) {

      //  $index 第几个

      //  $value == true 或者 $value == false

    }

}

--

5、发布订阅

( http://swoft.org/docs/2.x/zh-CN/redis/pub-sub.html )

(1)、发布 publish

描述: 将消息发布到频道。

//

///1/ 所需参数

channel: 要发布到的频道。

message: 消息。

//

///2/ 例子

Redis::publish('chan-1', 'hello, world!');  // 发送消息

Redis::publish('chan-2', 'hello, world2!'); // 发送消息

//

(2)、订阅 subscribe

描述: 订阅频道。

///1/ 所需参数

channels: 多个通道名称需要是一个数组。

callback: 回调函数接收3个参数(redis redis实例；chan 通道名称；msg 接受到的消息(mixed))。

function f($redis, $chan, $msg) {

    switch($chan) {

        case 'chan-1':

            ...

            break;

        case 'chan-2':

            ...

            break;

        case 'chan-3':

            ...

            break;

    }

}

// 订阅 3 个 频道

Redis::subscribe(['chan-1', 'chan-2', 'chan-3'], 'f');

//

(3)、按匹配模式订阅频道 pSubscribe

使用 psubscribe 方法可以订阅通配符频道，可以用来在所有频道上获取所有消息。

//

///1/ 参数

patterns 匹配规则数组。

callback 接受到消息回调的闭包(redis redis 实例；pattern 匹配的规则；chan 通道名称；msg 接收到的消息(mixed))。

//

///2/ 例子

Redis::psubscribe(['\*'], function ($redis, $pattern, $chan, $msg) {

    echo "Pattern: $pattern\n";

    echo "Channel: $chan\n";

    echo "Payload: $msg\n";

});

Redis::psubscribe(['users.\*'], function ($redis, $pattern, $chan, $msg) {

   echo "Pattern: $pattern\n";

   echo "Channel: $chan\n";

   echo "Payload: $msg\n";

});

//

发布订阅使用的 phpredis 原生的有可能过几秒会断开 socket 连接, 需要在开启发布订阅前加如下代码；

ini\_set('default\_socket\_timeout', -1);

// todo ...

Redis::psubscribe(['users.\*'], function ($redis, $pattern, $chan, $msg) {

   echo "Pattern: $pattern\n";

   echo "Channel: $chan\n";

   echo "Payload: $msg\n";

});

--

--

十一、国际化

--

--

0、概述

( http://swoft.org/docs/2.x/zh-CN/i18n/index.html )

(1)、使用场景

一般用于根据用户语言，需要输出不同的文案。如果没有国际化的封装，业务里面会有大量的判断，并且业务也不好维护，没法统一管理所有文案。

//

(2)、安装

composer require swoft/i18n

//

(3)、Git仓库

ub https://github.com/swoft-cloud/swoft-event

//

(4)、参与贡献

欢迎参与贡献，您可以，fork 我们的开发仓库 swoft/component；修改代码然后发起 PR；关于发起PR的注意事项。

--

1、配置参数

( http://swoft.org/docs/2.x/zh-CN/i18n/setting.html )

(1)、配置参数

使用 i18n 功能，一共需要三步，资源文件夹中导入语言文本；配置文件中配置相关参数；调用相关方法完成文本转换。

//

///1/ 导入文本

我们在 resource/language/ 文件夹下可创建语言组文件夹与文本文件，一个语言组中可以有多个模板文件，如下所示。

resource/language

|-- zh

    |-- default.php

    |-- msg.php

`-- en

    |-- default.php

    |-- msg.php

//

//// 文本格式：而文本格式是由关联数组组成。键值为文本，我们可以在其中插入参数，格式大致如下所述。

// ../en/default.php

return [

    // 文本中可用大括号注入参数

    'sayhello' => 'Hey {name}!',

    'saygoodbye' => 'Bye!',

];

//  ../en/msg.php

return [

    'sayhello' => "Wath's up! {name}",

    'saygoodbye' => 'See you tomorrow!',

];

// ../zh/default.php

return [

    'sayhello' => "早上好,{name}",

    'saygoodbye' => '再见',

];

// ../zh/msg.php

return [

    'sayhello' => "晚上好,{name}",

    'saygoodbye' => '明天见',

];

//

///2/ 相关配置

i18n相关功能配置非常简单，只需要在 app/bean.php 配置文件中，配置以下参数，即可开启国际化功能，参数描述见注释。

return [

    // .... 其他配置

    'i18n'  => [

        // 设置到文本资源目录

        'resoucePath' => '@resource/language/', // 结尾斜线必须

        // 设置默认文本文件夹名称

        // 未填写则默认 en 文件夹

        'defaultLanguage'   => 'en',

        // 设置默认文本文件名称

        // 未填写则默认 default.php

        'defualtCategory'   => 'default',

    ],

    // .... 其他配置

];

至此相关参数已配置完毕,下面就可以使用该功能了.

--

2、如何使用

( http://swoft.org/docs/2.x/zh-CN/i18n/usage.html )

//

///1/ 实用案例

// 语言转换仅仅至于要调用一个简单的方法, 下面是方法描述

/\*\*

 \* @param string $key       文本数组中,对应文本键值

 \* @param array  $params    注入到文本中的参数,以关联数组的形式

                            [

                                'param1' => 'str1',

                                'param2' => 'str2'

                            ]

 \* @param string $locale    资源文件夹下, 分组文件名称.

 \*/

\Swoft::t(string $key, array $params, string $locale): string

//

///2/ 实例演示

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Message\Request;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

use Swoft\Http\Server\Annotation\Mapping\RequestMethod;

/\*\*

 \* Test 用演示实例

 \*

 \* @since 1.0.0

 \*

 \* @Controller(perfix="test")

 \*/

class TestController

{

    // ...

    /\*\*

     \* i18n 示例

     \*

     \* @RequestMapping(route="i18n", method={RequestMethod::GET})

     \*

     \* @return array

     \*/

    public function i18n(Request $request):array

    {

        // 显示 language/en/default.php 文本模板:

        // 'sayhello' => 'Hey {name}!',

        $res['en'] = \Swoft::t('sayhello', ['name' => 'man']);

        // 更换模板可以使用 '.'

        // 显示 language/en/msg.php 模板

        $res['en-msg'] = \Swoft::t('msg.sayhello', ['name' => 'man'], 'en');

        // 显示 language/zh/default.php 模板

        $res['zh'] = \Swoft::t('sayhello', ['name' => '李华'], 'zh');

        // 显示 language/zh/msg.php 模板

        $res['zh-msg'] = \Swoft::t('msg.sayhello', ['name' => '李华'], 'zh');

        return $res;

    }

    // ...

}

?>

//

///3/ 结果输出

{

    "en": "Hey man!",

    "en-msg": "Wath's up! man",

    "zh": "早上好,李华",

    "zh-msg": "晚上好,李华"

}

--

--

十二、验证器

( http://swoft.org/docs/2.x/zh-CN/validator/index.html )

正常情况都需要对用户输入参数进行校验，此时就会用到验证器。验证器可以验证控制器中参数，也支持验证 Websocket 参数以及 RPC 参数验证，提供默认和自定义两种类型的验证器，还可添加自定义验证规则。

//

1、配置

( http://swoft.org/docs/2.x/zh-CN/validator/setting.html )

(1)、安装

使用验证器前，需要安装验证器组件，安装如下。

composer require swoft/validator

//

(2)、启用

成功安装好验证组件后，接下来需要启用验证器，这里以 Http-server 启用为例，其它一样(app/bean.php)。

return [

    // ......

    'httpDispatcher'    => [

        // ......

        'afterMiddlewares' => [

            \Swoft\Http\Server\Middleware\ValidatorMiddleware::class

        ]

        // ......

    ],

    // ......

];

2.0.5+ 验证器默认没有启动，需要开发者自己开启。

--

2、声明验证器

( http://swoft.org/docs/2.x/zh-CN/validator/statement.html )

一个验证器由多个验证条件组合，建议验证器按数据库表进行组合，这样可以充分的重复利用验证器里面的组合条件。

//

(1)、验证器

如下定义一个 TestValidator 验证器，由多个验证项（验证器条件）组成。

<?php declare(strict\_types=1);

namespace App\Validator;

use App\Annotation\Mapping\AlphaDash;

use Swoft\Validator\Annotation\Mapping\IsInt;

use Swoft\Validator\Annotation\Mapping\IsString;

use Swoft\Validator\Annotation\Mapping\Validator;

/\*\*

 \* Class TestValidator

 \*

 \* @since 2.0

 \*

 \* @Validator(name="TestValidator")

 \*/

class TestValidator

{

    /\*\*

     \* @IsString() //类型注解

     \* @Email() //条件注解

     \* @var string

     \*/

    protected $name = 'defualtName';

    /\*\*

     \* @IsInt(message="type must Integer")

     \*

     \* @var int

     \*/

    protected $type;

}

?>

//

//// @Validator

声明一个验证器，name 定义验证器的名称，方便引用，如果不定义默认就是类名全路径。

//

(2)、验证项

验证项是组成验证器的唯一条件，标记有类型注解的属性就是一个验证项，一个验证器可以有多个验证项。

|1>、属性的默认值就是参数的默认值，如果属性没有定义默认值，代表参数没有定义默认值且必须传递。

|2>、一个属性必须定义一个类型注解，否则不是一个验证项且对参数验证无效。

|3>、一个属性可以多个条件注解，按照定义顺序验证数据。

|4>、默认属性名称就是需要验证的参数名称，也可以通过类型注解的 name 参数映射需要验证的字段名称。

|5>、若验证不通过时，将会抛出 Swoft\Validator\Exception\ValidatorException 异常。

//

(2-1)、类型注解

( 一个属性必须定义一个类型注解，否则不是一个验证项且对参数验证无效。属性的默认值就是参数的默认值，如果属性没有定义默认值，代表参数没有定义默认值且必须传递 )

///1/ @IsArray

// 验证规则

验证参数值必须是数组,使用 is\_array()函数进行校验。

// 参数说明

name: 映射需要验证的字段名称，默认属性名称。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @IsArray(name="field", message="error message")。

//

///2/ @IsBool

// 验证规则

验证参数值必须是 bool 类型，注意字符串 true false，会验证成 bool 类型，其余数据将会使用is\_bool()函数进行验证。

// 参数说明

name: 映射需要验证的字段名称，默认属性名称。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @IsBool(name="field", message="error message")。

//

///3/ @IsFloat

// 验证规则

验证参数值必须是浮点数,使用filter\_var()函数进行验证。

// 参数说明

name: 映射需要验证的字段名称，默认属性名称。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @IsFloat(name="field", message="error message")。

//

///4/ @IsInt

// 验证规则

验证参数值必须是整数，使用filter\_var()函数进行验证。

// 参数说明

name: 映射需要验证的字段名称，默认属性名称。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @IsInt(name="field", message="error message")。

//

///5/ @IsString

// 验证规则

验证参数值必须是字符串，使用is\_string()函数进行验证。

// 参数说明

name: 映射需要验证的字段名称，默认属性名称。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @IsString(name="field", message="error message")。

//

(2-2)、条件注解

一个属性可以多个条件注解，按照定义顺序验证数据。

//

///1/ @AfterDate

// 验证规则

验证参数值必须在某个日期之后，参数支持 字符串时间戳、格式化日期字符串（只支持 Y-m-d H:i:s）、整型时间戳，可在@IsString 或 @IsInt 类型注解中使用。

// 参数说明

date: 要对比的日期值，只能是 Y-m-d H:i:s 格式。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @AfterDate(date="2019-01-01 00:00:00", message="error message")。

//

///2/ @Alpha

// 验证规则

验证参数值必须是 大写字母 或 小写字母。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Alpha(message="error message")。

//

///3/ @AlphaDash

// 验证规则

验证参数值必须是 大写字母 、 小写字母、数字、短横 -、下划线 \_。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @AlphaDash(message="error message")。

//

///4/ @AlphaNum

// 验证规则

验证参数值必须是 大写字母 、 小写字母、数字。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @AlphaNum(message="error message")。

//

///5/ @BeforeDate

// 验证规则

验证参数值必须在某个日期之前，支持 字符串时间戳、格式化日期字符串（只支持 Y-m-d H:i:s）、整型时间戳，可在@IsString 或 @IsInt 类型注解中使用。

// 参数说明

date: 要对比的日期值，只能是 Y-m-d H:i:s 格式。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @BeforeDate(date="2019-01-01 00:00:00", message="error message")。

//

///6/ @Chs

// 验证规则

验证参数值只能是 中文。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Chs(message="error message")。

//

///7/ @ChsAlpha

// 验证规则

验证参数值必须是 中文、大写字母 、 小写字母。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @ChsAlpha(message="error message")。

//

///8/ @ChsAlphaDash

// 验证规则

验证参数值必须是 中文、大写字母 、 小写字母、数字、短横 -、下划线 \_。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @ChsAlphaDash(message="error message")。

//

///9/ @ChsAlphaNum

// 验证规则

验证参数值必须是 中文、大写字母 、 小写字母、数字。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @ChsAlphaNum(message="error message")。

//

///10/ @Confirm

// 验证规则

验证参数值必须和另外一个字段参数值相同。

// 参数说明

name: 需要确认对比的字段名,在类型注解中设置过的 name 或者是默认的 属性名。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Confirm(name="field", message="error message")。

//

///11/ @Different

// 验证规则

验证参数值必须和另外一个字段参数值不同。

// 参数说明

name: 需要确认对比的字段名,在类型注解中设置过的 name 或者是默认的 属性名。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Different(name="field", message="error message")。

//

///12/ @GreaterThan

// 验证规则

验证参数值必须比另外一个字段参数值大，只支持 int 或 float, 字符串会被转化为 float 后进行对比。

// 参数说明

name: 需要确认对比的字段名,在类型注解中设置过的 name 或者是默认的 属性名。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @GreaterThan(name="field", message="error message")。

//

///13/ @LessThan

// 验证规则

验证参数值必须比另外一个字段参数值小，只支持 int 或 float, 字符串会被转化为 float 后进行对比。

// 参数说明

name: 需要确认对比的字段名,在类型注解中设置过的 name 或者是默认的 属性名。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @LessThan(name="field", message="error message")。

//

///14/ @Date

// 验证规则

验证参数值必须是日期格式，支持 字符串时间戳、格式化日期字符串（只支持 Y-m-d H:i:s）、整型时间戳，可在@IsString 或 @IsInt 类型注解中使用。注意由于时间戳的特殊性默认为一个整型 大于 PHP\_INT\_MIN , 小于 PHP\_INT\_MAX 常量的数值均为有效时间戳。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Date(message="error message")。

//

///15/ @DateRange

// 验证规则

验证参数值必须在某个日期范围之内（可以等于临界日期），支持 字符串时间戳、格式化日期字符串（只支持 Y-m-d H:i:s）、整型时间戳，可在@IsString 或 @IsInt 类型注解中使用。

// 参数说明

start: 要对比的开始日期值，只能是 Y-m-d H:i:s 格式。

end: 要对比的结束日期值，只能是 Y-m-d H:i:s 格式。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @DateRange(start="2019-01-01 00:00:00",end="2019-01-01 00:00:00", message="error message")。

//

///16/ @Dns

// 验证规则

验证参数值必须是一个具有有效 DNS 记录域名或者ip，使用 checkdnsrr() 函数校验。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Dns(message="error message")。

//

///17/ @Email

// 验证规则

验证参数值格式必须为邮箱

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Email(message="error message")。

//

///8/ @Enum

// 验证规则

验证参数值必须在枚举数组里面。

// 参数说明

values: 枚举数组集合

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Enum(values={1,2,3},message="error message")。

//

///9/ @File

// 验证规则

验证此参数的值必须是文件，可以是单个文件，也可以是表单数组上传的多个文件。注意文件上传后文件域的获取需要通过 Swoft\Http\Message\Request 对象去获取。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @File(message="error message")。

//

///10/ @FileMediaType

// 验证规则

使用此条件前必须使用 @File 规则为基础。 验证每个上传的文件 mediaType 类型,支持表单数组，批量文件。

// 参数说明

mediaType: mediaType类型数组。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @FileMediaType(mediaType={"image/gif","image/png"},message="error message")。

//

///11/ @FileSize

// 验证规则

使用此条件前必须使用 @File 规则为基础。 验证每个上传的文件尺寸大小（单位 byte 字节）,支持表单数组，批量文件。

// 参数说明

size: 文件尺寸大小，单位 byte。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @FileSize(size=1024,message="error message")。

//

///12/ @FileSuffix

// 验证规则

使用此条件前必须使用 @File 规则为基础。 验证每个上传的文件后缀名,支持表单数组，批量文件。

// 参数说明

suffix: 后缀名数组。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @FileSuffix(suffix={"png","jpg"},message="error message")。

//

///13/ @Ip

// 验证规则

验证参数值必须是个IP类型。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Ip(message="error message")。

//

///14/ @Length

// 验证规则

验证参数值长度限制。

// 参数说明

min: 最小值(包含当前值)。

max: 最大值(包含当前值)。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Length(min=1,max=4,message="error message")。

//

///15/ @Low

// 验证规则

验证参数值必须是小写字母。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Low(message="error message")。

//

///16/ @Max

// 验证规则

最大值验证，必须是整数。

// 参数说明

value 最大值(包含当前值)

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Max(value=5,message="error message")。

//

///17/ @Min

// 验证规则

最小值验证。

// 参数说明

value 最小值(包含当前值)

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Min(value=5,message="error message")。

//

///18/ @Mobile

// 验证规则

手机号验证。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Mobile(message="error message")。

//

///19/ @NotEmpty

// 验证规则

参数值不能为空验证。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @NotEmpty(message="error message")。

//

///20/ @NotInEnum

// 验证规则

验证参数值必须不在枚举数组中。

// 参数说明

values 枚举数组集合。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @NotInEnum(values={1,2,3},message="error message")。

//

///21/ @NotInRange

// 验证规则

验参数值必须不在范围内。

// 参数说明

min: 最小值(包含当前值)。

max: 最大值(包含当前值)。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @NotInRange(min=1,max=5,message="error message")。

//

///22/ @Pattern

// 验证规则

正则表达式验证。

// 参数说明

regex: 正则表达式。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Pattern(regex="/^1\d{10}$/", message="error message")。

//

///23/ @Range

// 验证规则

参数值范围验证。

// 参数说明

min: 最小值(包含当前值)。

max: 最大值(包含当前值)。

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Range(min=1,max=5,message="error message")。

//

///24/ @Upper

// 验证规则

验证参数值必须是大写字母。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Range(message="error message")。

//

///25/ @Url

// 验证规则

验证参数值必须是有效的URL格式，使用 filter\_var()函数验证。

// 参数说明

message: 验证失败时的错误提示，若不设置则默认使用框架内置的。

使用示例: @Url(message="error message")。

暂时官方提供了这些条件验证，如果有其它需求可以根据 自定义验证器规则( http://swoft.org/docs/2.x/zh-CN/validator/customer-rule.html ) 此章节内容自行添加验证规则。

--

3、自定义验证器

( http://swoft.org/docs/2.x/zh-CN/validator/user-validator.html )

(1)、声明验证器

声明验证器很简单。定一个类实现 Swoft\Validator\Contract\ValidatorInterface 接口；实现 validate 方法，里面可以定义自己的业务验证器逻辑；使用 @Validator 注解标记这是一个验证器，此注解使用和功能和之前介绍完全一样。

<?php declare(strict\_types=1);

namespace App\Validator;

use Swoft\Validator\Annotation\Mapping\Validator;

use Swoft\Validator\Contract\ValidatorInterface;

use Swoft\Validator\Exception\ValidatorException;

/\*\*

 \* Class CustomerValidator

 \*

 \* @since 2.0

 \*

 \* @Validator(name="userValidator")

 \*/

class CustomerValidator implements ValidatorInterface

{

    /\*\*

     \* @param array $data

     \* @param array $params

     \*

     \* @return array

     \* @throws ValidatorException

     \*/

    public function validate(array $data, array $params): array

    {

        $start = $data['start'] ?? null;

        $end = $data['end'] ?? null;

        if ($start === null && $end === null) {

            throw new ValidatorException('Start time and end time cannot be empty');

        }

        if ($start > $end) {

            throw new ValidatorException('Start cannot be greater than the end time');

        }

        return $data;

    }

}

?>

//

///1/ 验证方法详细介绍

public function validate(array $data, array $params): array

$data 用户输入参数，通过对应的解析器，已解析成数组。

$params 传递给验证器的参数，后续章节详细介绍。

验证成功返回验证处理后的数据 , 如果验证失败，抛出 Swoft\Validator\Exception\ValidatorException 异常，其它由框架处理。如果验证器里面需要修改参数值，可以直接修改，修改后获取值的地方会得到新的值。

--

4、自定义验证器规则

( http://swoft.org/docs/2.x/zh-CN/validator/customer-rule.html )

之前我们介绍过了如何使用自定义验证器，但是有些时候我们想要扩展我们的验证器规则，而非重新自定义一个验证器，对于这种场景的应用，我们也能够比较方便的添加我们自己的验证器规则，请阅读下方详细步骤。(Available: >= v2.0.3)

//

(1)、声明验证器

我们在 声明验证器( http://swoft.org/docs/2.x/zh-CN/validator/statement.html ) 这一节中声明的验证器基础上，添加一个 password 字段，使用一个我们自定义的验证规则来验证该字段。

/\*\*

 \* @AlphaDash(message="Passwords can only be alphabet, numbers, dashes, underscores")

 \*

 \* @var string

 \*/

protected $password;

如上我们就添加好了一个 password 字段，并且使用了一个 @AlphaDash() 的验证规则，该规则就是我们接下来要自定义的规则，它的功能是校验该字段的格式，使其只能是 字母,数字,-,\_。

//

(2)、声明规则注解

注意 我们强烈建议你按照 应用结构(http://swoft.org/docs/2.x/zh-CN/quick-start/project-skeleton.html) 中的建议，将自定义的注解定义到 App/Annotation 路径中。

///1/ 声明注解命令

<?php declare(strict\_types=1);

namespace App\Annotation\Mapping;

use Doctrine\Common\Annotations\Annotation\Attribute;

use Doctrine\Common\Annotations\Annotation\Attributes;

/\*\*

 \* Class AlphaDash

 \*

 \* @since 2.0

 \*

 \* @Annotation

 \* @Attributes({

 \*     @Attribute("message",type="string")

 \* })

 \*/

class AlphaDash

{

    /\*\*

     \* @var string

     \*/

    private $message = '';

    /\*\*

     \* @var string

     \*/

    private $name = '';

    /\*\*

     \* StringType constructor.

     \*

     \* @param array $values

     \*/

    public function \_\_construct(array $values)

    {

        if (isset($values['value'])) {

            $this->message = $values['value'];

        }

        if (isset($values['message'])) {

            $this->message = $values['message'];

        }

        if (isset($values['name'])) {

            $this->name = $values['name'];

        }

    }

    /\*\*

     \* @return string

     \*/

    public function getMessage(): string

    {

        return $this->message;

    }

    /\*\*

     \* @return string

     \*/

    public function getName(): string

    {

        return $this->name;

    }

}

?>

//

@Annotation 声明这个类是一个注解命令。

@Attributes 声明注解参数集合。

@Attribute 声明注解具体的参数，name 参数的名字；type 参数值的类型。

// 类属性方法说明

$message 就是我们使用该注解时传入的提示。例如 @IsString(message="该字段必须是字符串") $name 就是字段的名字，为空的话则默认就是属性名 例如 @IsString(name="user\_name") $value 则是我们需要传递给验证规则的一些数据，若无需传参则可以不用定义。例如 @Enum(values=[1,2,3]) getMessage(),getName() 方法必须存在。用来获取$message 和 $name。

//

///2/ 声明注解解析

至此，我们定义好了注解命令，但是注解命令要想能够执行，则还需要定义一个注解命令的解析器，下方就是一个注解解析器，需要继承 Swoft\Annotation\Annotation\Parser\Parser 类。

<?php declare(strict\_types=1);

namespace App\Annotation\Parser;

use ReflectionException;

use Swoft\Annotation\Annotation\Mapping\AnnotationParser;

use Swoft\Annotation\Annotation\Parser\Parser;

use App\Annotation\Mapping\AlphaDash;

use Swoft\Validator\Exception\ValidatorException;

use Swoft\Validator\ValidatorRegister;

/\*\*

 \* Class AlphaDashParser

 \*

 \* @AnnotationParser(annotation=AlphaDash::class)

 \*/

class AlphaDashParser extends Parser

{

    /\*\*

     \* @param int $type

     \* @param object $annotationObject

     \*

     \* @return array

     \* @throws ReflectionException

     \* @throws ValidatorException

     \*/

    public function parse(int $type, $annotationObject): array

    {

        if ($type != self::TYPE\_PROPERTY) {

            return [];

        }

        //向验证器注册一个验证规则

        ValidatorRegister::registerValidatorItem($this->className, $this->propertyName, $annotationObject);

        return [];

    }

}

?>

//

@AnnotationParser 声明要解析的注解命令。

// parse()

由于我们继承了 Swoft\Annotation\Annotation\Parser\Parser , 而它有声明了一个 Swoft\Annotation\Annotation\Parser\ParserInterface 接口,而这个方法正是 ParserInterface 这个接口所定义的一个必须由我们来实现的一个接口。 其实这里就是我们要处理的业务逻辑了，Swoft 解析到一个注解命令后，就会执行这个注解所对应的解析器中的 parse() 这个方法。

///3/ 声明一个验证规则

经过之前的步骤我们已经定义好了验证规则的注解以及它的解析器，但是我们还没有定义我们的具体的验证规则，所以接下来，我们将声明我们具体的验证规则， 其实很简单，我们只需要实现一个 Swoft\Validator\Contract\RuleInterface 接口就可以了。

<?php declare(strict\_types=1);

namespace App\Validator\Rule;

use App\Annotation\Mapping\AlphaDash;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Validator\Contract\RuleInterface;

use Swoft\Validator\Exception\ValidatorException;

/\*\*

 \* Class AlphaDashRule

 \*

 \* @Bean(AlphaDash::class)

 \*/

class AlphaDashRule implements RuleInterface

{

    /\*\*

     \* @param array $data

     \* @param string $propertyName

     \* @param object $item

     \* @param null $default

     \*

     \* @return array

     \* @throws ValidatorException

     \*/

    public function validate(array $data, string $propertyName, $item, $default = null): array

    {

        $message = $item->getMessage();

        if (!isset($data[$propertyName]) && $default === null) {

            $message = (empty($message)) ? sprintf('%s must exist!', $propertyName) : $message;

            throw new ValidatorException($message);

        }

        $rule = '/^[A-Za-z0-9\-\\_]+$/';

        if (preg\_match($rule, $data[$propertyName])) {

            return [$data];

        }

        $message = (empty($message)) ? sprintf('%s must be a email', $propertyName) : $message;

        throw new ValidatorException($message);

    }

}

?>

//// @Bean

由于验证器内部是通过 Bean 容器来获取得到我们的验证规则的，代码如下。

$rule = BeanFactory::getBean($itemClass);//这里通过容器拿到了我们的验证规则

$data = $rule->validate($data, $propName, $item, $default);

所以这里我们就要使用 @Bean 来注册我们的验证规则，名字就是和我们的注解命令相同。

//

//// validate()

这是 RuleInterface 接口中规定要实现的方法，到了这里其实就是写我们具体的验证规则了。

array $data 待验证的所有数据；string $propertyName 需要验证的字段名；$item 注解类的对象；$default 字段的默认值。至此我们已经定义好了一个验证器规则。

--

5、控制器中使用

( http://swoft.org/docs/2.x/zh-CN/validator/controller-validator.html )

如果想在控制器中使用验证器很简单，只需要是一个注解 @Validate 就行。一个 action 可以定义多个 @Validate 使用多个验证器；多个验证器按照配置顺序验证。

//// 如下定义一个 ValidatorController, 同时使用默认验证器和自定义验证器以及我们自定义的验证规则。

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Http\Message\Request;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

use Swoft\Validator\Annotation\Mapping\Validate;

/\*\*

 \* Class ValidatorController

 \*

 \* @Controller()

 \*/

class ValidatorController

{

    /\*\*

     \* 验证TestValidator验证器中的所有已定义字段

     \*

     \* @RequestMapping()

     \* @Validate(validator="TestValidator")

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function validateAll(Request $request): array

    {

        return $request->getParsedBody();

    }

    /\*\*

     \* 仅验证TestValidator验证器中的 type 字段

     \*

     \* @RequestMapping()

     \* @Validate(validator="TestValidator",fields={"type"})

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function validateType(Request $request): array

    {

        return $request->getParsedBody();

    }

    /\*\*

     \* 仅验证TestValidator验证器中的 password 字段 password字段使用的是自定义的验证规则。

     \*

     \* @RequestMapping()

     \* @Validate(validator="TestValidator",fields={"password"})

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function validatePassword(Request $request): array

    {

        return $request->getParsedBody();

    }

    /\*\*

     \* 使用userValidator自定义验证器

     \*

     \* @RequestMapping()

     \* @Validate(validator="userValidator")

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function validateCustomer(Request $request): array

    {

        return $request->getParsedBody();

    }

}

?>

$request->getParsedBody() 所有解析数据

$request->parsedBody('key', 'default') 指定 KEY 解析数据

$request->getParsedQuery() 所有解析的 Query 参数

$request->parsedQuery('key', 'default') 指定 KEY 解析数据(>=2.0.2)

//

(1)、@Validate

validator 指定验证器名称；fields 指定验证器里面验证的字段，这样可以高效的重复使用验证器；type 默认 body，ValidateType::GET 验证 GET 请求 query 参数；params 自定义验证器使用，传递给自定义验证器的参数。

注意 $request->getParsedBody() 获取的请求数据，是已经通过验证器修改的数据。验证器可以支持表单、请求 body 数据验证，但是 body 验证需要定义对应的数据解析器，框架默认提供 JSON/XML 类型数据解析器，详细介绍，请参考 Http Server 章节。

--

6、非注解使用

( http://swoft.org/docs/2.x/zh-CN/validator/anywhere-validator.html )

注解方式引用和使用验证器是有限制，只支持在 Http server/ Rpc server /Websocket server 等特定位置使用，在实际业务开发中，其它地方也会涉及参数的验证。 非注解和注解方式都是引用相同的验证器，一个验证器可以多个位置，多种方式使用，大大的减少了代码的重复成本。

//

(1)、全局方法

function validate(array $data, string $validatorName, array $fields = [], array $userValidators = []): array

全局函数使用，当验证器失败会抛出 Swoft\Validator\Exception\ValidatorException 异常。

$data 需要验证的数据，必须是数组 KV 格式。

$validatorName 使用的验证器( @Validator() 注解标记的 )。

$fields 需要验证的字段，为空验证器所有字段。

$userValidators 同时使用的自定义验证器，支持两种格式。

//

(2)、使用示例

///1/ 所有参数验证

use Swoft\Validator\Annotation\Mapping\Validator;

$data = [

    'email' => 'swoft@xx'

]

$result = validate($data, Validator::class);

//

///2/ 指定字段验证

use Swoft\Validator\Annotation\Mapping\Validator;

$data = [

    'email' => 'swoft@xx'

];

$result = validate($data, Validator::class, ['email']);

//

///3/ 同时使用自定义验证器

use Swoft\Validator\Annotation\Mapping\Validator;

$data = [

    'start'  => 12,

    'end'    => 16,

];

$result = validate($data, Validator::class, [], ['testUserValidtor']);

//

///4/ 同时使用自定义验证器且传递参数

use Swoft\Validator\Annotation\Mapping\Validator;

$data = [

    'start'  => 12,

    'end'    => 16,

    'params' => [1, 2]

];

$users = [

    'testUserValidtor' => [1, 2]

];

$result = validate($data, Validator::class, [], $users);

--

--

十三、日志

( http://swoft.org/docs/2.x/zh-CN/log/index.html )

框架日志由控制台日志和应用日志组成，控制台日志一般用于调试打印，应用日志用于记录开发者的业务日志和框架运行日志。 无论是控制台日志还是应用日志都提供了很多灵活的参数，方便开发者。

swoft/log 基于著名的php日志库monolog扩展而来。你可以很方便的添加自定义handler或processor。

//

1、控制台日志

( http://swoft.org/docs/2.x/zh-CN/log/clog.html )

Swoft提供简便的控制台日志使用，便于在开发时打印调试信息。

//

(1)、配置

启动应用里面 (app\Application.php) 重写父类方法，可以覆盖配置控制台日志参数。

namespace App;

use Swoft\SwoftApplication;

/\*\*

 \* Class Application

 \*

 \* @since 2.0

 \*/

class Application extends SwoftApplication

{

    public function getCLoggerConfig(): array

    {

        return [

            'name'    => 'swoft',

            'enable'  => true,

            'output'  => true,

            'levels'  => 'info,error',

            'logFile' => ''

        ];

    }

}

// 选项说明

name 名称。

enable 是否开启。

output 是否打印到控制台。

levels 输入日志的级别，为空全部输出，具体日志级别配置值，可以引用 Logger::NOTICE/...。

logFile 控制台日志默认打印到控制台，也可以配置路径，同时写到指定文件。

提示：swoft 2.0.3 levels 修改成字符串，方便开发者覆盖框架默认配置。

//

(2)、使用

每个日志级别方法都可以传递参数，底层是一个 sprintf() 函数封装。

特殊的：debug 日志级别，需要开启 SWOFT\_DEBUG 才会显示。

框架内置不同级别不同颜色。

// debug

CLog::debug('debug');

// info

CLog::info('info');

// warning

CLog::warning('warning');

// error

CLog::error('error');

//

// 2019/05/12-07:02:57 [DEBUG] Swoft\Processor\ConsoleProcessor:handle(33) debug

// 2019/05/12-07:02:57 [INFO] Swoft\Processor\ConsoleProcessor:handle(33) info

// 2019/05/12-07:02:57 [WARNING] Swoft\Processor\ConsoleProcessor:handle(33) warning

// 2019/05/12-07:02:57 [ERROR] Swoft\Processor\ConsoleProcessor:handle(33) error

//

(3)、关闭信息

默认情况下，启动时会打印一些启动信息到控制台。如果你的 .env 开启了 SWOFT\_DEBUG=1 将会看到更多详细的启动与加载信息。

//

//// 如果你想关闭这些信息，可以在 app/Application 添加。

public function getCLoggerConfig(): array

{

    $config = parent::getCLoggerConfig();

    // Disable print console log

    $config['enable'] = false;

    return $config;

}

修改保存后，重启swoft，可以看到不会有任何信息输出了。

--

2、应用日志

( http://swoft.org/docs/2.x/zh-CN/log/log.html )

(1)、配置

// 在bean.php 里面配置应用日志的参数

return [

    'lineFormatter'      => [

        'format'     => '%datetime% [%level\_name%] [%channel%] [%event%] [tid:%tid%] [cid:%cid%] [traceid:%traceid%] [spanid:%spanid%] [parentid:%parentid%] %messages%',

        'dateFormat' => 'Y-m-d H:i:s',

    ],

    'noticeHandler'      => [

        'class'     => FileHandler::class,

        'logFile'   => '@runtime/logs/notice-%d{Y-m-d}.log',  // 2.0.6 支持日志按时间切割

        'formatter' => \bean('lineFormatter'),

        'levels'    => 'notice,info,debug,trace',

    ],

    'applicationHandler' => [

        'class'     => FileHandler::class,

        'logFile'   => '@runtime/logs/error.log',

        'formatter' => \bean('lineFormatter'),

        'levels'    => 'error,warning',

    ],

    'logger'             => [

        'flushRequest' => false, //是否每个请求结束输出日志，默认 false；

        'enable'       => false, //是否开启日志，默认 false；

        'handlers'     => [

            'application' => \bean('applicationHandler'),

            'notice'      => \bean('noticeHandler'),

        ],

    ]

];

此配置也是框架默认的配置文件，把应用日志按日志级别分别写到两个不同的文件里面。swoft 2.0.3 levels 修改成字符串，方便开发者覆盖框架默认配置。

//

///1/ 日志格式

lineFormatter 配置日志格式：format 日志输到文件格式；dateFormat 日志输出时间格式。

//

///2/ 日志处理器

noticeHandler 和 applicationHandler 处理器，应用日志可以配置多个处理器，处理器可以把日志输出到文件、邮箱、第三方系统。

//

// 配置详细参数

class 配置采用哪种类型的，框架默认提供文件，用户可以自己扩展其它类型。

logFile 输出日志文件路径，支持别名( 2.0.6 支持日志按时间切割，格式： %d{Y-m-d} ， { } 中的格式适用 PHP date format )。

formatter 日志输出使用日志格式，就是之前配置的日志格式。

levels 支持日志输出的日志级别。

//

///3/ 日志配置

// logger 日志配置

name 名称。

flushInterval 日志输出频率默认 1。

flushRequest 是否每个请求结束输出日志，默认 false。

enable 是否开启日志，默认 false。

json 是否 JSON 格式输出，默认 false。

items 日志其它数据，配置是一个数组集合。

//

(2)、使用

// 开发者直接通过 Log 类静态方法，打印日志。应用日志根据不同的需求分为如下不同的日志记录。

emergency

debug

alert

info

warning

error

//

// Tag start

Log::profileStart('tagName');

//

Log::debug('this %s log', 'debug');

Log::info('this %s log', 'info');

Log::warning('this %s log', 'warning');

Log::error('this %s log', 'error');

Log::alert('this %s log', 'alert');

Log::emergency('this %s log', 'emergency');

//

// Pushlog

Log::pushLog('key', 'value');

Log::pushLog('key', ['value']);

Log::pushLog('key', 'value');

//

// Tag end

Log::profileEnd('tagName');

//

// Counting

Log::counting('mget', 1, 10);

Log::counting('mget', 2, 10);

//

// 2019-05-11 06:57:27 [info] [swoft] [request] [tid:4] [cid:4] [traceid:] [spanid:] [parentid:] trace[LogController.php:29,App\Http\Controller\LogController->test] info message

// 2019-05-11 06:57:27 [debug] [swoft] [request] [tid:4] [cid:4] [traceid:] [spanid:] [parentid:] trace[LogController.php:35,App\Http\Controller\LogController->test] this debug log

// 2019-05-11 06:57:27 [info] [swoft] [request] [tid:4] [cid:4] [traceid:] [spanid:] [parentid:] trace[LogController.php:36,App\Http\Controller\LogController->test] this info log

// 2019-05-11 06:57:27 [warning] [swoft] [request] [tid:4] [cid:4] [traceid:] [spanid:] [parentid:] trace[LogController.php:37,App\Http\Controller\LogController->test] this warning log

// 2019-05-11 06:57:27 [error] [swoft] [request] [tid:4] [cid:4] [traceid:] [spanid:] [parentid:] trace[LogController.php:38,App\Http\Controller\LogController->test] this error log

// 2019-05-11 06:57:27 [notice] [swoft] [request] [tid:4] [cid:5] [traceid:] [spanid:] [parentid:] [8.52(ms)] [27(MB)] [/log/test] [key=value key=["value"] key=value] profile[tagName=5.02(ms)/2] counting[mget=3/20]

//

日志级别可以根据自己的业务定义不同的含义，debug 日志级别只有开启 APP\_DEBUG 默认才会输出. notice 日志级别，每个请求结束框架自动追加上的一条日志，该条日志会记录这次请求，框架和业务运行的详细信息，比如数据库操作记录、缓存操作记录以及请求信息。日志方法可以像 sprintf() 函数一样传递参数，底层自动封装。

//

///1/ 除常用的日志级别，框架还提供 pushLog profile

pushLog 用于记录请求执行过程中的参数，notice 日志里面可以查看。

profile 使用的时，需要定义一个标记，标记开始和结束为止，该方法可以记录标记之间运行时间，notice 日志里面可以查看。

--

--

第五篇 微服务

--

--

一、服务治理

( http://swoft.org/docs/2.x/zh-CN/ms/govern/index.html )

微服务，我们建议使用 Service mesh 方式，比如 Istio/Envoy 框架，业务和服务治理分开，但是 Swoft 也为开发者快速构建微服务提供了一套组件。

服务注册与发现；

熔断器；

限流；

配置中心；

Swoft 版本必须 2.0.3+。

--

1、注册与发现 ( 服务注册与发现 )

微服务治理过程中，经常会涉及注册启动的服务到第三方集群，比如 consul / etcd 等等，本章以 Swoft 框架中使用 swoft-consul 组件，实现服务注册与发现为例。

//

(1)、服务注册

无论是 http / rpc / ws 服务，启动的时候只需监听 SwooleEvent::START 事件，即可把启动的服务注册到第三方集群。

//

///1/ 注册服务

// 本章这里以启动 http server 注册服务为例

<?php declare(strict\_types=1);

namespace App\Listener;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Consul\Agent;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Http\Server\HttpServer;

use Swoft\Log\Helper\CLog;

use Swoft\Server\Swoole\SwooleEvent;

/\*\*

 \* Class RegisterServiceListener

 \*

 \* @since 2.0

 \*

 \* @Listener(event=SwooleEvent::START)

 \*/

class RegisterServiceListener implements EventHandlerInterface

{

    /\*\*

     \* @Inject()

     \*

     \* @var Agent

     \*/

    private $agent;

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        /\* @var HttpServer $httpServer \*/

        $httpServer = $event->getTarget();

        $service = [

            'ID'                => 'swoft',

            'Name'              => 'swoft',

            'Tags'              => [

                'http'

            ],

            'Address'           => '127.0.0.1',

            'Port'              => $httpServer->getPort(),

            'Meta'              => [

                'version' => '1.0'

            ],

            'EnableTagOverride' => false,

            'Weights'           => [

                'Passing' => 10,

                'Warning' => 1

            ]

        ];

        // Register

        $this->agent->registerService($service);

        CLog::info('Swoft http register service success by consul!');

    }

}

?>

//

// 详细流程

$event->getTarget() 获取的对象是启动服务的对象，比如启动的 http 拿到的就是 Swoft\Http\Server\HttpServer 对象。

target 对象里面可以获取启动服务的所有信息包括配置信息。

根据服务配置信息和业务实际情况，注册服务信息到 consul。

提示：必须使用 Swoole\Coroutine\Scheduler 否则无法注册服务，swoole 4.4+ 支持该功能。

//

///2/ 取消服务

服务启动注册服务，服务关闭或者退出则需要取消服务注册，此时这里和注册一样监听一个 SwooleEvent::SHUTDOWN 事件即可，本章这里还是以 取消 Http server 服务为例。

<?php declare(strict\_types=1);

namespace App\Listener;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Consul\Agent;

use Swoft\Event\Annotation\Mapping\Listener;

use Swoft\Event\EventHandlerInterface;

use Swoft\Event\EventInterface;

use Swoft\Http\Server\HttpServer;

use Swoft\Server\Swoole\SwooleEvent;

/\*\*

 \* Class DeregisterServiceListener

 \*

 \* @since 2.0

 \*

 \* @Listener(SwooleEvent::SHUTDOWN)

 \*/

class DeregisterServiceListener implements EventHandlerInterface

{

    /\*\*

     \* @Inject()

     \*

     \* @var Agent

     \*/

    private $agent;

    /\*\*

     \* @param EventInterface $event

     \*/

    public function handle(EventInterface $event): void

    {

        /\* @var HttpServer $httpServer \*/

        $httpServer = $event->getTarget();

        $this->agent->deregisterService('swoft');

    }

}

?>

//

// 详细流程

$event->getTarget() 获取的对象是启动服务的对象，比如启动的 http 拿到的就是 Swoft\Http\Server\HttpServer 对象。

target 对象里面可以获取启动服务的所有信息包括配置信息。

根据服务配置信息和业务实际情况，取消已经支持的服务。

提示：必须使用 Swoole\Coroutine\Scheduler 否则无法注册服务，swoole 4.4+ 支持该功能。

//

(2)、服务发现

本章这里以 Rpc client 为例，通过第三方集群 consul 下发可用的服务列表。

//

///1/ 首先定义服务提供者

<?php declare(strict\_types=1);

namespace App\Common;

use ReflectionException;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Consul\Agent;

use Swoft\Consul\Exception\ClientException;

use Swoft\Consul\Exception\ServerException;

use Swoft\Rpc\Client\Client;

use Swoft\Rpc\Client\Contract\ProviderInterface;

/\*\*

 \* Class RpcProvider

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class RpcProvider implements ProviderInterface

{

    /\*\*

     \* @Inject()

     \*

     \* @var Agent

     \*/

    private $agent;

    /\*\*

     \* @param Client $client

     \*

     \* @return array

     \* @throws ReflectionException

     \* @throws ContainerException

     \* @throws ClientException

     \* @throws ServerException

     \* @example

     \* [

     \*     'host:port',

     \*     'host:port',

     \*     'host:port',

     \* ]

     \*/

    public function getList(Client $client): array

    {

        // Get health service from consul

        $services = $this->agent->services();

        $services = [

        ];

        return $services;

    }

}

?>

//

// 详细流程

实现 Swoft\Rpc\Client\Contract\ProviderInterface 接口。

根据 $client 参数获取当前 RPC 服务信息。

根据服务信息，到第三方集群(consul) 里面查询可用服务。

返回一个规定格式数组。

此类，必须用 @Bean 标记为一个 bean 对像。

提示：db / redis 也是支持这种方式发现可用的服务，但是需要实现其相应的接口。

//

// 现在我们有了服务提供者，接下来将其配置到对应的 RPC 服务上面。

// app/bean.php

return [

    'user'           => [

      'class'   => ServiceClient::class,

      'host'    => '127.0.0.1',

      'port'    => 18307,

      'setting' => [

          'timeout'         => 0.5,

          'connect\_timeout' => 1.0,

          'write\_timeout'   => 10.0,

          'read\_timeout'    => 0.5,

      ],

      'packet'  => bean('rpcClientPacket'),

      'provider' => bean(RpcProvider::class)

    ],

    'user.pool'      => [

      'class'  => ServicePool::class,

      'client' => bean('user')

    ],

];

这里在 user 服务上，通过 provider 参数注入了一个服务提供者 RpcProvider::class (bean 名称)。

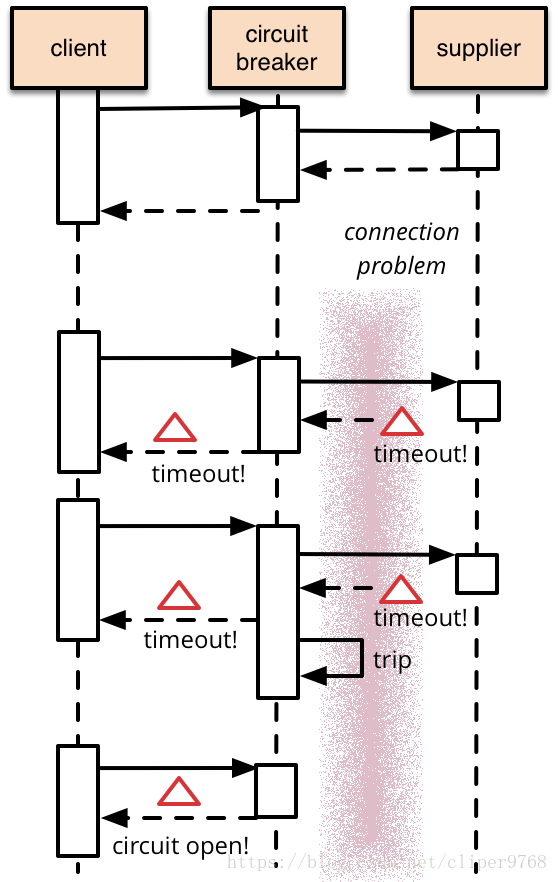
--

2、服务熔断

(0)、熔断器

在分布式环境下，特别是微服务结构的分布式系统中， 一个软件系统调用另外一个远程系统是非常普遍的。这种远程调用的被调用方可能是另外一个进程，或者是跨网路的另外一台主机, 这种远程的调用和进程的内部调用最大的区别是，远程调用可能会失败，或者挂起而没有任何回应，直到超时。更坏的情况是， 如果有多个调用者对同一个挂起的服务进行调用，那么就很有可能的是一个服务的超时等待迅速蔓延到整个分布式系统，引起连锁反应， 从而消耗掉整个分布式系统大量资源。最终可能导致系统瘫痪。断路器（Circuit Breaker）模式就是为了防止在分布式系统中出现这种瀑布似的连锁反应导致的灾难。

( 基本模式图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ms/breaker\_base.png )



// 上图是断路器（Curcuit Breaker）的结构，它有两个基本状态（close和open）和一个基本trip动作

close状态下，client向supplier发起的服务请求，直接无阻碍通过断路器，supplier的返回值直接由断路器交回给client。

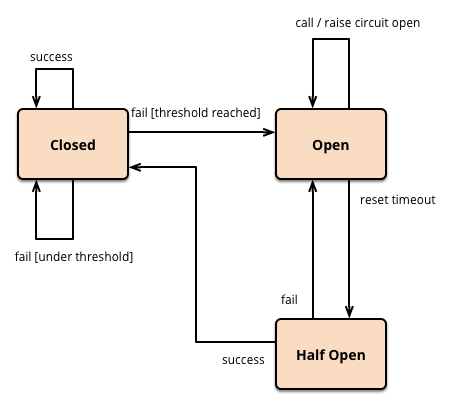
open状态下，client向supplier发起的服务请求后，断路器不会将请求转到supplier, 而是直接返回client, client和supplier之间的通路是断的。

trip：在close状态下，如果supplier持续超时报错，达到规定的阀值后，断路器就发生trip，之后断路器状态就会从close进入open。

//

///2/ 扩展模式

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ms/breaker\_ext.png )



基本的断路器模式下，保证了断路器在open状态时，保护supplier不会被调用， 但我们还需要额外的措施可以在supplier恢复服务后，可以重置断路器。一种可行的办法是断路器定期探测supplier的服务是否恢复， 一但恢复， 就将状态设置成close。断路器进行重试时的状态为半开（half-open）状态。

Swoft 熔断器底层采用扩展模式设计，开发者必须先熟悉熔断器状态变化，这样有利于对熔断器的理解。

//

(1)、安装

// 使用熔断器前必须安装 swoft-breaker 组件，安装方式如下所示

composer require swoft/breaker

//

(2)、配置

// 熔断器不使用配置也是可以使用的，但是有些业务场景需要全局配置熔断器，可以参考如下配置。

// app/bean.php

return [

    'breaker' => [

         'timeout' => 3,

    ]

]

//

// 详细参数

timeout 超时时间。

failThreshold 连续失败多少次状态切换阀门。

sucThreshold 连续成功多少次状态切换阀门。

retryTime 熔断器由开启状态到半开状态尝试切换时间。

提示：全局配置会被 @Breaker 上面的局部注解覆盖。

//

(3)、使用

熔断器的使用相当简单且功能强大，使用一个 @Breaker 注解即可，Swoft 中的熔断是针对于类里面的方法熔断，只要方法里面没有抛出异常就说明是成功访问的，所以 @Breaker 注解可以在任何 bean对象方法上面使用。

<?php declare(strict\_types=1);

namespace App\Model\Logic;

use Exception;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Breaker\Annotation\Mapping\Breaker;

/\*\*

 \* Class BreakerLogic

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class BreakerLogic

{

    /\*\*

     \* @Breaker(fallback="funcFallback")

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function func(): string

    {

        // Do something

        throw new Exception('Breaker exception');

    }

    /\*\*

     \* @Breaker()

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function func2(): string

    {

        // Do something

        return 'func2';

    }

    /\*\*

     \* @return string

     \*/

    public function funcFallback(): string

    {

        return 'funcFallback';

    }

    /\*\*

     \* @Breaker()

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function unFallback(): string

    {

        // Do something

        throw new Exception('Breaker exception');

    }

    /\*\*

     \* @Breaker(fallback="loopFallback")

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function loop(): string

    {

        // Do something

        throw new Exception('Breaker exception');

    }

    /\*\*

     \* @Breaker(fallback="loopFallback2")

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function loopFallback(): string

    {

        // Do something

        throw new Exception('Breaker exception');

    }

    /\*\*

     \* @Breaker(fallback="loopFallback3")

     \*

     \* @return string

     \* @throws Exception

     \*/

    public function loopFallback2(): string

    {

        // Do something

        throw new Exception('Breaker exception');

    }

    /\*\*

     \* @return string

     \*/

    public function loopFallback3(): string

    {

        return 'loopFallback3';

    }

}

?>

//

// 这里示例定义了几种使用场景

函数正常没有一次抛出 (func2)。

函数抛出异常通过降级函数返回数据 (func)。

定义熔断器没有定义降级函数 (unFallback)。

循环执行熔断器 (loop)。

//

// 循环执行熔断器这里详细讲解下流程

执行 loop 函数，抛出异常，执行 loopFallback 降级函数。

loopFallback2 降级函数，抛出异常，但是定义了熔断器和降级函数，所以继续执行 loopFallback2 降级函数。

loopFallback2 降级函数，此时这里正常返回降级的数据。

//

提示：`fallback` 函数必须和 `@Breaker` 标记的函数完全一样除了名称不一样且在同一个类里面。

//

(4)、注解

//// @Breaker

熔断器就用了一个 @Breaker 注解，该注解，必须定义在 bean 对象里面。

//

// 标记方法开启熔断器，如下参数详解

fallback 降级函数，必须和 @Breaker 标记的函数完全一样除了名称不一样且在同一个类里面。

sucThreshold 连续成功多少次状态切换阀门。

failThreshold 连续失败多少次状态切换阀门。

timeout 超时时间。

retryTime 熔断器由开启状态到半开状态尝试切换时间。

注意：注解上面的配置会覆盖熔断器全局配置。

--

3、服务限流

( http://swoft.org/docs/2.x/zh-CN/ms/govern/limiter.html )

限流是对稀缺资源访问时，比如秒杀，抢购的商品时，来限制并发和请求的数量，从而有效的进行削峰并使得流量曲线平滑。限流的目的是对并发访问和并发请求进行限速，或者一个时间窗口内请求进行限速从而来保护系统，一旦达到或超过限制速率就可以拒绝服务，或者进行排队等待等。

//

(1)、限流算法

///1/ 计数器

采用计数器实现限流有点简单粗暴，一般我们会限制一秒钟的能够通过的请求数，比如限流qps为100，算法的实现思路就是从第一个请求进来开始计时，在接下去的1s内，每来一个请求，就把计数加1，如果累加的数字达到了100，那么后续的请求就会被全部拒绝。等到1s结束后，把计数恢复成0，重新开始计数。

这种实现方式，相信大家都知道有一个弊端：如果我在单位时间1s内的前10ms，已经通过了100个请求，那后面的990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象”。

//

///2/ 漏桶

为了消除"突刺现象"，可以采用漏桶算法实现限流，漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏斗，当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速地流出。不管上面流量多大，下面流出的速度始终保持不变。

不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，没来得及处理的请求就先放在桶里，既然是个桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。

( 漏桶图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ms/limiter\_leaky.png )

这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。

//

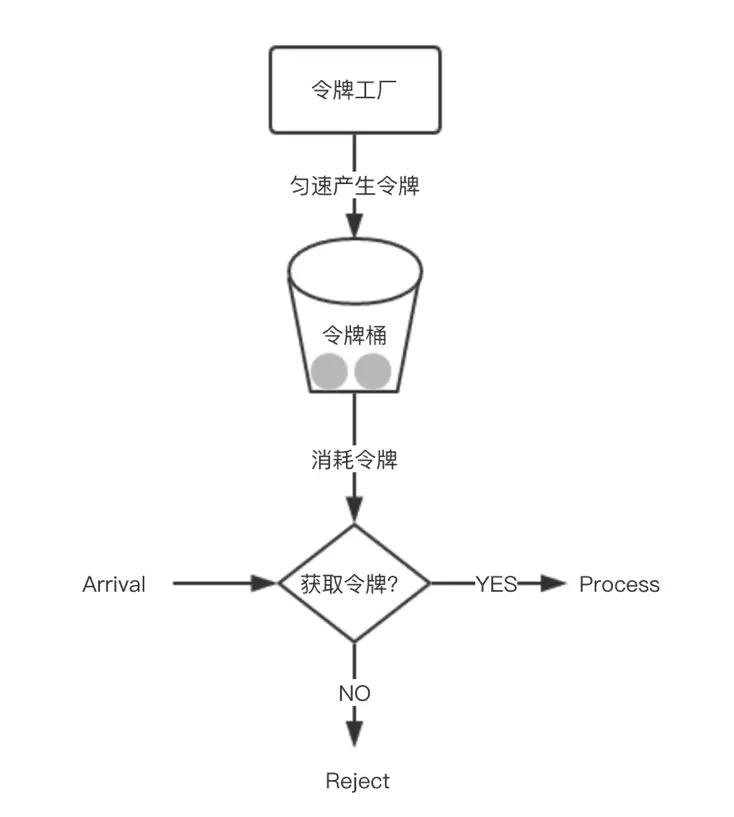
///3/ 令牌桶

某种意义上讲，令牌桶算法是对漏桶算法的一种改进，桶算法能够限制请求调用的速率，而令牌桶算法能够在限制调用的平均速率的同时还允许一定程度的突发调用。

在令牌桶算法中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。

放令牌这个动作是持续不断的进行，如果桶中令牌数达到上限，就丢弃令牌，所以就存在这种情况，桶中一直有大量的可用令牌，这时进来的请求就可以直接拿到令牌执行，比如设置qps为100，那么限流器初始化完成一秒后，桶中就已经有100个令牌了，这时服务还没完全启动好，等启动完成对外提供服务时，该限流器可以抵挡瞬时的100个请求。所以，只有桶中没有令牌时，请求才会进行等待，最后相当于以一定的速率执行。

( 令牌桶图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/ms/limiter\_token.png )



Swoft 限流器底层采用的是令牌桶算法，底层依赖于 Redis 实现分布式限流。

//

(2)、令牌桶实现原理

///1/ 对于令牌桶中令牌的产生一般有两种做法

|1>、一种解法是，开启一个定时任务，由定时任务持续生成令牌。这样的问题在于会极大的消耗系统资源，如，某接口需要分别对每个用户做访问频率限制，假设系统中存在6W用户，则至多需要开启6W个定时任务来维持每个桶中的令牌数，这样的开销是巨大的。

|2>、第二种解法是延迟计算，定义一个 resync 函数。该函数会在每次获取令牌之前调用，其实现思路为，若当前时间晚于nextFreeTicketMicros，则计算该段时间内可以生成多少令牌，将生成的令牌加入令牌桶中并更新数据。这样一来，只需要在获取令牌时计算一次即可。

Swoft 采用的是第二种，当每次获取令牌时，先执行resync来更新令牌桶中令牌的数量，从而达到异步产生令牌的目的。下面开始详解讲解，如何使用 swoft-limiter 组件，进行开发。

//

(3)、安装

// 如上已经详解讲解了限流器实现原理，使用限流器前，必须安装 swoft-limiter 组件，安装命令如下。

composer require swoft/limiter

//

(4)、配置

// 限流器不使用配置也是可以使用的，但是有些业务场景需要全局配置限流器，可以参考如下配置

return [

    'rateLimiter' => [

        'class'      => RateLimter::class,

        'rateLimter' => bean('redisRateLimiter'),

    ]

];

// 详细参数

name 限流器名称，默认 swoft:limiter。

rate 允许多大的请求访问，请求数/秒。

max 最大的请求数。

default 初始化请求数。

提示：全局配置会被 @Limiter 注解的局部配置覆盖。

//

// 缓存限速信息也是可以配置的，配置如下

return [ 'redisRateLimiter' => [ 'pool' => 'redis.pool' ]; ];

pool 指定使用的缓存连接池名称，默认使用框架默认的连接池。使用限速器，一定要配置安装 redis 组件，且配置可用的 redis 缓存。

//

(5)、使用

///1/ 控制器限速

// 本节以常见的限速场景为例

<?php declare(strict\_types=1);

namespace App\Http\Controller;

use Swoft\Http\Message\Request;

use Swoft\Http\Server\Annotation\Mapping\Controller;

use Swoft\Http\Server\Annotation\Mapping\RequestMapping;

use Swoft\Limiter\Annotation\Mapping\RateLimiter;

/\*\*

 \* Class LimiterController

 \*

 \* @since 2.0

 \*

 \* @Controller(prefix="limiter")

 \*/

class LimiterController

{

    /\*\*

     \* @RequestMapping()

     \* @RateLimiter(key="request.getUriPath()")

     \*

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function requestLimiter(Request $request): array

    {

        $uri = $request->getUriPath();

        return ['requestLimiter', $uri];

    }

    /\*\*

     \* @RequestMapping()

     \* @RateLimiter(rate=20, fallback="limiterFallback")

     \*

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function requestLimiter2(Request $request): array

    {

        $uri = $request->getUriPath();

        return ['requestLimiter2', $uri];

    }

    /\*\*

     \* @RequestMapping()

     \* @RateLimiter(key="request.getUriPath()~':'~request.query('id')")

     \*

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function paramLimiter(Request $request): array

    {

        $id = $request->query('id');

        return ['paramLimiter', $id];

    }

    /\*\*

     \* @param Request $request

     \*

     \* @return array

     \*/

    public function limiterFallback(Request $request): array

    {

        $uri = $request->getUriPath();

        return ['limiterFallback', $uri];

    }

}

?>

key 这里支持 symfony/expression-language 表达式，可以实现很多复杂的功能，详细文档( http://www.symfonychina.com/doc/current/components/expression\_language/syntax.html ), 如果被限速会抛出 Swoft\Limiter\Exception\RateLImiterException 异常。key 表达式内置 CLASS(类名) 和 METHOD(方法名称) 两个变量，方便开发者使用。

//

///2/ 方法限速

// Swoft 限速器不仅可以限流控制器，也可以限制任何 bean 里面的方法，可以控制方法的访问速率。这里以下面使用示例详解

<?php declare(strict\_types=1);

namespace App\Model\Logic;

use Swoft\Bean\Annotation\Mapping\Bean;

use Swoft\Limiter\Annotation\Mapping\RateLimiter;

/\*\*

 \* Class LimiterLogic

 \*

 \* @since 2.0

 \*

 \* @Bean()

 \*/

class LimiterLogic

{

    /\*\*

     \* @RateLimiter(fallback="limiterFallback")

     \*

     \* @return array

     \*/

    public function limiter(): array

    {

        // Do something

        return [];

    }

    /\*\*

     \* @RateLimiter(key="requestBean.getName('name')")

     \*

     \* @param RequestBean $requestBean

     \*

     \* @return array

     \*/

    public function limiterParams(RequestBean $requestBean): array

    {

        // Do something

        return [];

    }

    /\*\*

     \* @return array

     \*/

    public function limiterFallback(): array

    {

        return [];

    }

}

?>

limiter 方法，根据类名+方法名称限速且定义了一个降级函数。

limtierParams 方法，根据调用 RequestBean 对象的方法，返回值作为限流 key。

fallback指定的函数必须和limiter一样，包括名称和参数、返回值，并且必须在同一个类里面。 如果被限速会抛出 `Swoft\Limiter\Exception\RateLImiterException` 异常。key 表达式内置 `CLASS`(类名) 和 `METHOD`(方法名称) 两个变量，方便开发者使用。

//

(6)、注解

限流器注解很简单，只涉及一个 @RateLimiter 注解。

//

///1/ @RateLimiter

// 标记方法，开启限流策略，详细参数如下

name 缓存前缀。

rate 允许多大的请求访问，请求数/秒。

key 限流 key, 支持 symfony/expression-language 表达式，key 表达式内置 CLASS(类名) 和 METHOD(方法名称) 两个变量，方便开发者使用。

max 最大的请求数。

default 初始化请求数。

fallback 降级函数，和 breaker 一样。

--

4、配置中心

( http://swoft.org/docs/2.x/zh-CN/ms/govern/config.html )

配置中心主要就是把配置集中化管理，方便统一和维护。本章以 Apollo 为例，从远端配置中心拉取配置以及安全重启服务。如果对 Apollo 不熟悉，可以先看 Swoft 扩展 Apollo 组件以及阅读 Apollo 官方文档。

//

///1/ 配置中心使用流程

编写本地 agent 监听配置的变化，如果有变更，修改本地配置文件。

重启业务对应的服务。

提示：本地 agent 一定要比服务先启动，否则服务启动，没法获取最新的配置信息。

//

(1)、使用

本章以 Swoft 中使用 apollo 为例，当 apollo 配置变更后，重启服务(http-server / rpc-server/ ws-server)。如下是一个 agent 例子。

//

///1/ 声明 agent

<?php declare(strict\_types=1);

namespace App\Console\Command;

use ReflectionException;

use Swoft\Apollo\Config;

use Swoft\Apollo\Exception\ApolloException;

use Swoft\Bean\Annotation\Mapping\Inject;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Co;

use Swoft\Console\Annotation\Mapping\Command;

use Swoft\Console\Annotation\Mapping\CommandMapping;

use Swoft\Http\Server\HttpServer;

use Swoft\Log\Helper\CLog;

use Swoft\Rpc\Server\ServiceServer;

use Swoft\WebSocket\Server\WebSocketServer;

use Throwable;

/\*\*

 \* Class AgentCommand

 \*

 \* @since 2.0

 \*

 \* @Command("agent")

 \*/

class AgentCommand

{

    /\*\*

     \* @Inject()

     \*

     \* @var Config

     \*/

    private $config;

    /\*\*

     \* @CommandMapping(name="index")

     \*/

    public function index(): void

    {

        $namespaces = [

            'application'

        ];

        while (true) {

            try {

                $this->config->listen($namespaces, [$this, 'updateConfigFile']);

            } catch (Throwable $e) {

                CLog::error('Config agent fail(%s %s %d)!', $e->getMessage(), $e->getFile(), $e->getLine());

            }

        }

    }

    /\*\*

     \* @param array $data

     \*

     \* @throws ContainerException

     \* @throws ReflectionException

     \*/

    public function updateConfigFile(array $data): void

    {

        foreach ($data as $namespace => $namespaceData) {

            $configFile = sprintf('@config/%s.php', $namespace);

            $configKVs = $namespaceData['configurations'] ?? '';

            $content   = '<?php return ' . var\_export($configKVs, true) . ';';

            Co::writeFile(alias($configFile), $content, FILE\_NO\_DEFAULT\_CONTEXT);

            CLog::info('Apollo update success！');

            /\*\* @var HttpServer $server \*/

            $server = bean('httpServer');

            $server->restart();

//            /\*\* @var ServiceServer $server \*/

//            $server = bean('rpcServer');

//            $server->restart();

//            /\* @var WebSocketServer $server \*/

//            $server = bean('wsServer');

//            $server->restart();

        }

    }

}

?>

这里声明了一个本地 agent，它监听 apollo 远程配置变更，如果有变更，回调函数，同时返回最新的配置信息，根据业务实际情况重新修改配置文件，再重启相应的服务。

//

///2/ 启动 agent

php bin/swoft agent:index

agent 启动，可以后台守护进程启动，防止挂掉。

//

///3/ 启动服务

// 以启动 Http server 为例

/usr/local/php/bin/php /data/www/swoft/bin/swoft http:start -d

//

启动命令的路径，必须是绝对路径 (/data/www/swoft/bin/swoft) 且后台运行模式 (-d)，服务的启动必须在 agent 之后，否则无法获取最新配置信息。本章只是一个简单的使用例子，开发者可以根据自己的实际业务情况，监听 apollo 配置变更，生成配置文件，重启服务。

--

--

第六篇 工具

--

--

一、Swoft CLI

0、Swoft CLI

// Swoft CLI 是一个独立的命令行应用包，提供了一些内置的工具方便开发者使用。

|1>、将一个swoft应用打包成 phar 包；

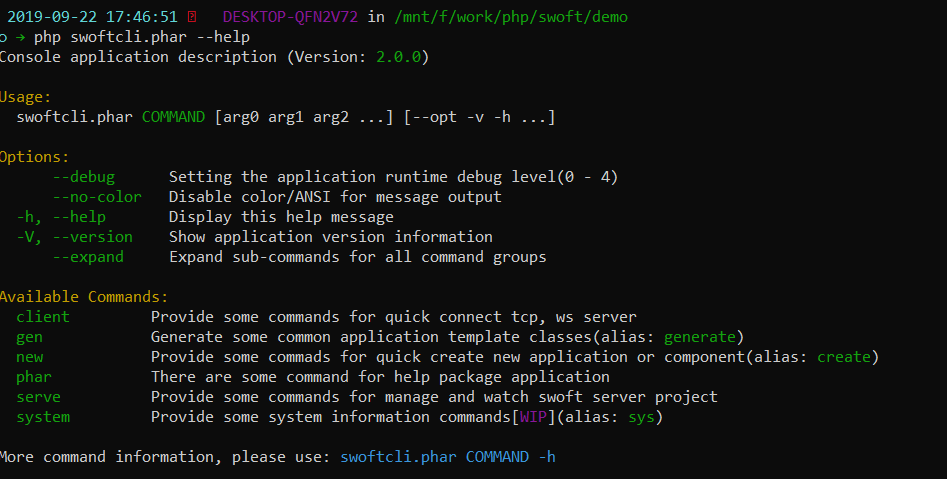
|2>、监视用户swoft项目的文件更改并自动重新启动服务器；

提示：后续会根据用户需要增加更多的帮助工具，欢迎用户提供意见和贡献代码。swoft-cli 是基于 swoft 2.0 构建的应用，运行使用同样需要swoole。

//

(1)、运行预览

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-home.png )



//

(2)、仓库

github https://github.com/swoft-cloud/swoft-cli

//

(3)、参与贡献

您可以 fork 仓库修改然后发起 PR。

--

1、下载安装

( http://swoft.org/docs/2.x/zh-CN/tool/swoftcli/install.html )

安装 swoftcli 非常简单。我们提供已经打包好的phar放在GitHub上。

//

(1)、下载

你需要从 swoft-cli 的 GitHub Releases( https://github.com/swoft-cloud/swoft-cli/releases ) 下载打包好的 swoftcli.phar。

注意：需要将下面命令里的 {VERSION} 替换为指定的版本。当然你也可以直接通过浏览器下载。

// 下载 https://github.com/swoft-cloud/swoft-cli/releases/download/v0.0.6/swoftcli.phar

wget https://github.com/swoft-cloud/swoft-cli/releases/download/{VERSION}/swoftcli.phar

// 检查包是否可用

php swoftcli.phar -V

php swoftcli.phar -h

//

(2)、全局使用

// 如果你需要在任何地方都可以直接使用 swoftcli

// move to ENV path:

mv swoftcli.phar /usr/local/bin/swoftcli

chmod a+x /usr/local/bin/swoftcli

// check

swoftcli -V

//

(3)、构建

// 如果你需要从最新的代码构建phar包

git clone https://github.com/swoft-cloud/swoft-cli

cd swoft-cli

composer install

// build

php -d phar.readonly=0 bin/swoftcli phar:pack -o=swoftcli.phar

--

2、自动重启服务

( http://swoft.org/docs/2.x/zh-CN/tool/swoftcli/hot-restart.html )

swoft 2.0 在内置组件中去除了自动重启功能，由新的独立工具 swoft-cli 来提供。需要注意的是，2.0 里面是重启整个服务而不是像 1.0 一样只reload工作进程。

命令：serve:run (也可以直接使用别名 run)。

//

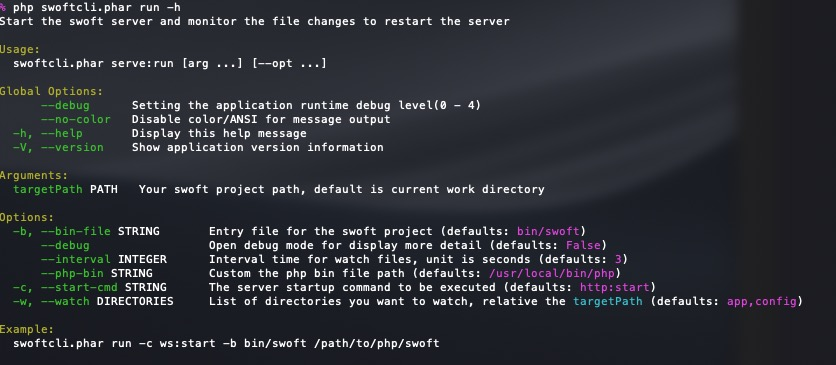
(1)、查看可用选项

php swoftcli.phar run -h

// 如果已放到了全局PATH里，可以直接这样使用

swoftcli run -h

( 运行图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/hot-restart-help.jpg )



//

///1/ 选项列表

-b, --bin-file 指定swoft应用的入口文件，默认是 bin/swoft。

--interval 监控文件的间隔时间，默认 3 秒钟检查一次。

--php-bin 指定你的php可执行文件，默认会自动从全局path中寻找php。

-c, --start-cmd 指定server启动命令，默认是 http:start (启动http server)。

-w, --watch 指定要监控的目录，相对于应用目录。默认监控 app,config 里的文件变动。

//

///2/ 参数列表

targetPath 仅有一个参数，指定要运行的swoft应用所在目录，默认为当前目录。

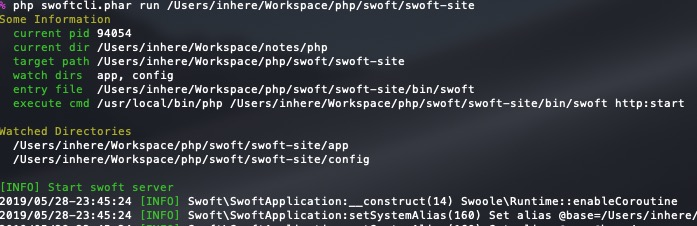
//

(2)、使用示例

swoftcli run -c ws:start -b bin/swoft /path/to/php/swoft

// 运行成功后你可以看到如下的信息

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/run-hot-restart.jpg )



当有文件发生变动时，swoft-cli 就会自动的重新启动应用。

警告⚠️ 使用 swoftcli 监控server开发时，不能将server配置为后台运行，不然swoftcli会错误地认为server意外退出了，导致重复启动。

--

3、生成应用类文件

( http://swoft.org/docs/2.x/zh-CN/tool/swoftcli/generate-app-classs.html )

自swoftcli v0.1.0 起，支持通过内置命令快速地创建一个应用类。

//

///1/ 支持创建

console command class

event listener class

task class

crontab task class

http controller class

http middleware class

rpc controller class

rpc middleware class

tcp controller class

websocket module class

websocket controller class

and more ...

//

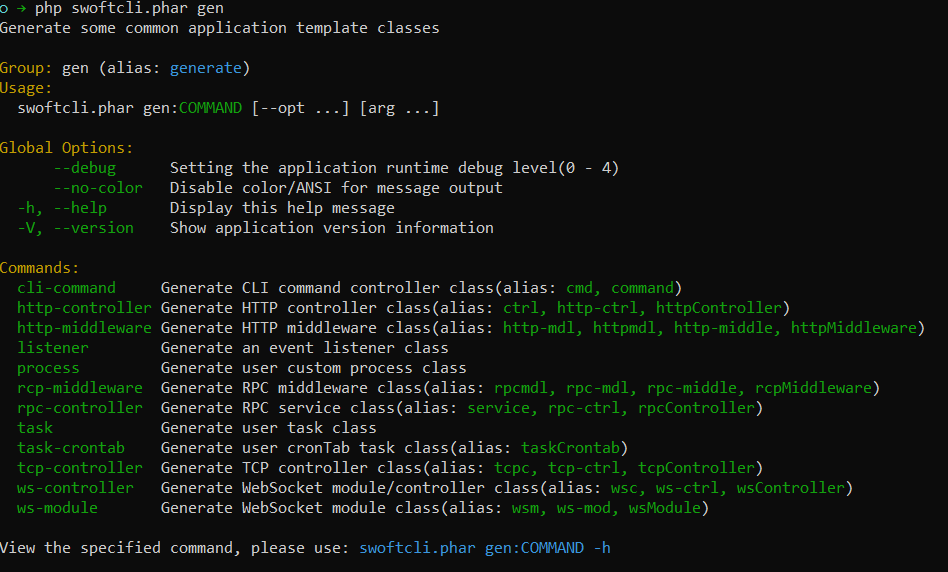
(1)、查看命令组

php swoftcli.phar gen

// or

php swoftcli.phar gen --help

( 运行图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-gen.png )



///1/ 生成http controller

php swoftcli.phar gen:http-ctrl user --prefix /users

//

///2/ 生成http middleware

php swoftcli.phar gen:http-mdl demo

//

///3/ 生成websocket module

php swoftcli.phar gen:ws-mod chat --prefix /chat

//

///4/ 生成websocket controller

php swoftcli.phar gen:ws-ctrl chat

--

4、创建新应用或组件

( http://swoft.org/docs/2.x/zh-CN/tool/swoftcli/create-app-or-component.html )

自swoftcli v0.1.0 起，支持通过内置命令快速的创建一个新的应用骨架，或者创建一个新的组件骨架结构。

//

(1)、查看命令组

php swoftcli.phar new

// or use alias

php swoftcli.phar create

//

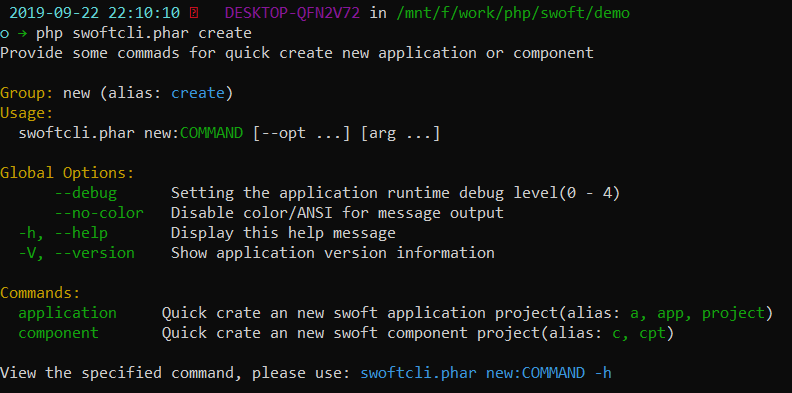
// 命令组说明

命令组 new (组别名 create)，通过任意一个都可以访问到它。

拥有两个子命令，application 和 component 分别用于创建新的应用和新的组件结构。

//

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-new.png )



//

(2)、创建新应用

创建新应用是通过拉取github上已存在的模板项目仓库，因此您可以轻松自定义符合自己需求的模板。

创建新应用是通过拉取github上已存在的模板项目仓库，因此您可以轻松自定义符合自己需求的模板。swoft默认提供了5个模板仓库，方便用户根据需要拉取不同的骨架结构。可以在下面的命令帮助中看到有哪几个默认骨架。

//

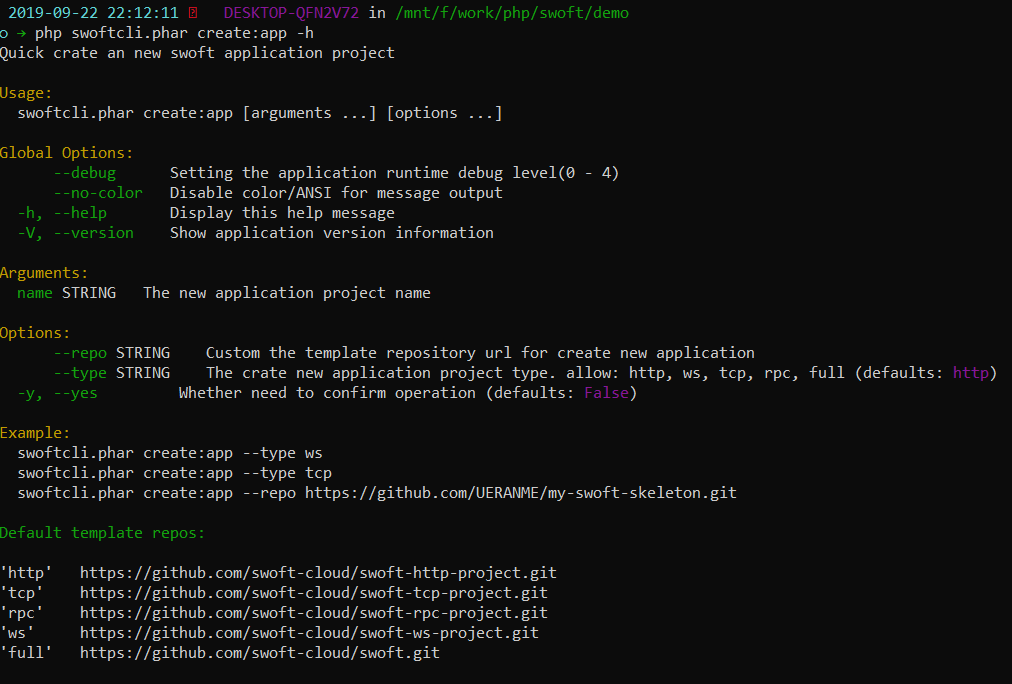
///1/ 查看命令

php swoftcli.phar create:app -h

// or use alias

php swoftcli.phar create:app

( 运行图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-new-app.png )



//

///2/ 命令使用说明

// 命令参数

name string，设置新项目名称，同时也是作为新应用的目录名，必须。

// 命令选项

--repo string 自定义设置模板仓库的git地址，可以是 UERANME/REPO 或者 完整url地址。

--type string 从默认的提供的5个模板仓库里选择一个来作为源仓库，默认是 http。

提示：YOUR\_APP\_NAME 指的是你的新项目名称，同时也是作为新应用的目录名。

//

// 直接使用 create:app YOUR\_APP\_NAME，默认使用 swoft-http-project 模板仓库。 如果需要ws/tcp/rpc等模板仓库作为基础模板，可以如下指定 type 选项。

php swoftcli.phar create:app --type ws YOUR\_APP\_NAME

php swoftcli.phar create:app --type tcp YOUR\_APP\_NAME

//

// 如果你需要对模板做一些完全的自定义，那你就可以在自己的github创建需要的模板仓库，然后使用如下命令来使用。

php swoftcli.phar create:app --repo UERANME/REPO YOUR\_APP\_NAME

//

// 使用的完整的git仓库地址；这时不限于从github拉取，即也可以从自己的git服务拉取来初始化一个新应用。

php swoftcli.phar create:app --repo https://github.com/UERANME/REPO.git YOUR\_APP\_NAME

//

///3/ 使用命令创建新应用

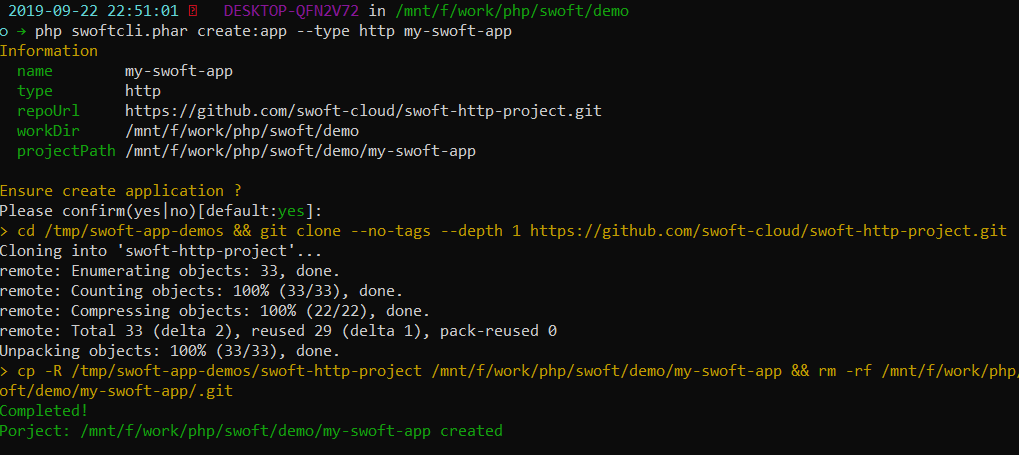
// 我们来试试创建新应用的命令

php swoftcli.phar create:app --type http my-swoft-app

//

// 我们可以看到如下信息

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-new-app-example.png )



//

(3)、创建新组件

swoft 也提供了简单快捷的命令来创建一个新的组件骨架。骨架结构非常简单，包含 composer.json README.md AutoLoader.php 等基础文件。

//

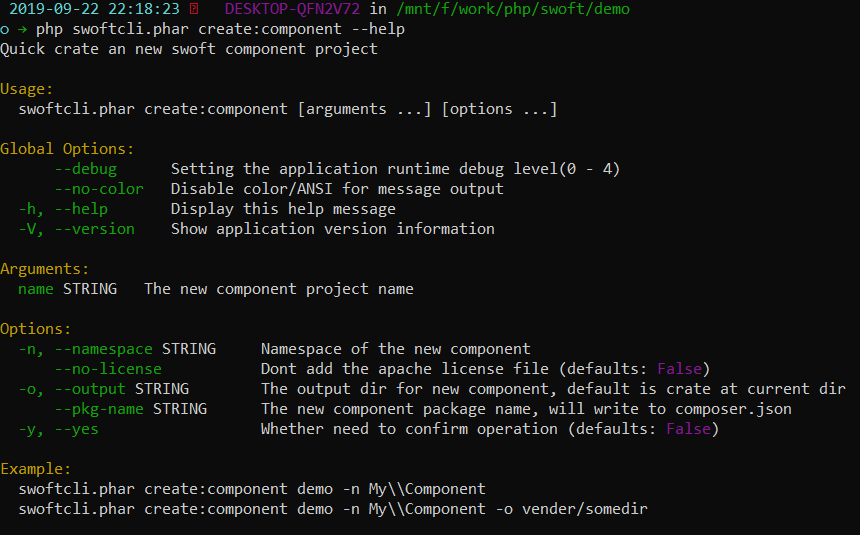
///1/ 查看命令

php swoftcli.phar create:component --help

// or use alias

php swoftcli.phar create:c

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-new-cpt.png )



//

///2/ 命令使用说明

// 命令参数

name string 设置新组件的名称，同时也是作为组件目录名，必须。

//

// 命令选项

-n, --namespace string 设置组件的命名空间，设置后会自动写入到 composer.json 的PSR4加载。

--no-licence bool 是否添加 LICENSE 文件，默认会添加。

-o, --output string 组件创建输出路径；不设置则创建到当前目录下。

--pgk-name string 组件的包名，会自动写入到 composer.json，为空则会根据当前用户名加上组件名组成。

//

///3/ 使用命令创建新组件

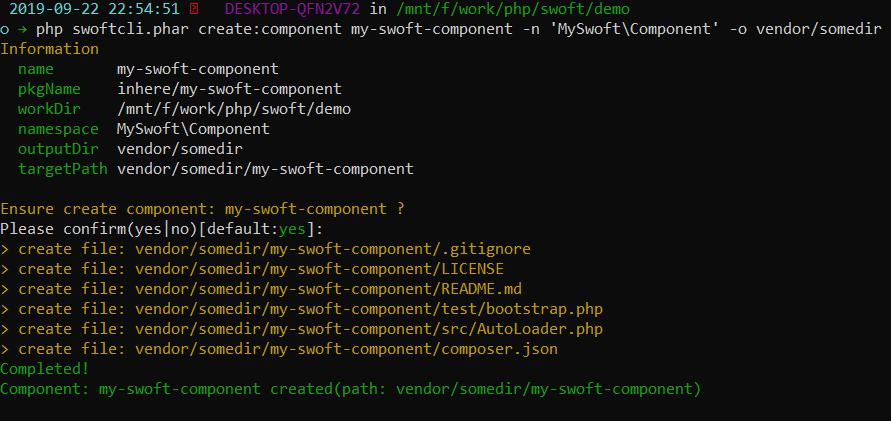
// 我们来试试创建一个新的组件结构。

php swoftcli.phar create:component my-swoft-component -n 'MySwoft\Component' -o vendor/somedir

//

// 我们可以看到如下信息

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/swoftcli-new-cpt-example.png )



//

5、打包应用为PHAR

(0)、打包Phar

swoft-cli 工具提供了一个简单的命令，可用于将一个php应用（不限于swoft）打包成一个Phar包。要使用此功能，请确保 phar 扩展是启用的。通常情况下，这个扩展在php里是默认启用的。

//

(1)、构建Phar

php -d phar.readonly=0 swoftcli phar:pack -o=myapp.phar

//

///1/ 可用选项(通过帮助查看)

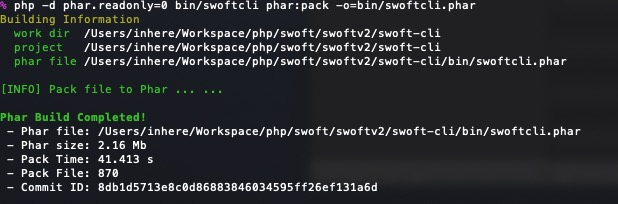
-c, --config 指定打包用的配置文件，默认读取当前目录下的 phar.build.inc。

--dir 要打包的应用目录，默认是当前目录。

-o, --output 打包后输出的文件名称，默认是 app.phar。

注意：运行时必须在前面为php加上选项 -d phar.readonly=0。

( 运行效果图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/swoftcli/pack-phar.jpg )



//

//// phar 打包配置

phar 打包需要一个配置文件，可以配置一些需要打包的目录、文件等。通常放置于项目根目录下。

<?php

/\*\*

 \* The is a config file for compile phar package.

 \* @var \Swoft\Console\Advanced\PharCompiler $compiler

 \*/

// config

$compiler

    // 删除代码注释。打包swoft 应用需要配合 setStripFilter()

    ->stripComments(true)

    ->setShebang(true)

    // 需要打包的文件后缀，内置已经包含了 .php

    ->addSuffix(['.tpl', '.json'])

    // 排除打包目录，排除一些无用的目录，可以有效减少输出文件大小

    ->addExcludeDir([

        'test',

        'doc',

        'docs',

        'tests',

        'Tests',

        'example',

        'runtime',

        'swoole-ide-helper',

    ])

    // 指定添加的文件

    ->addFile([

        'LICENSE',

        'composer.json',

        'README.md',

        'bin/bootstrap.php',

    ])

    ->setCliIndex('bin/swoft') // 应用入口

    // ->setWebIndex('web/index.php')

    // ->setVersionFile('config/config.php')

;

// 那些目录需要去除注释。Swoft 相关都不能去除注释

$stripDirs = [

    'psr/',

    'nikic/',

    'monolog/',

    'symfony/',

    'toolkit/',

];

$compiler->setStripFilter(function ($file) use ($stripDirs) {

    /\*\* @var \SplFileInfo $file \*/

    $path = $file->getPathname();

    foreach ($stripDirs as $dir) {

        if (\strpos($path, $dir)) {

            return true;

        }

    }

    return false;

});

?>

//

(2)、展开Phar包

// 将一个打包过的 phar 展开，类似于解压缩。

swoftcli phar:unpack myapp.phar

--

--

二、Devtool

--

0、开发者工具

( http://swoft.org/docs/2.x/zh-CN/tool/devtool/index.html )

swoft 提供了一些内置的工具方便开发者使用。

//

(1)、安装组件

//// 绝大部分工具都放置在组件 swoft/devtool 中

composer require swoft/devtool

//

(2)、CLI帮助命令

提供了一些CLI帮助命令，以便用户快速生成一些基本class。生成后的类文件只需稍加调整就可使用。

运行环境检查。

提供database table实体类生成。

提供基本的class模板文件生成。

//

(3)、方便的web UI

server, application 信息显示。

已注册的路由信息(http,websocket,rpc)。

简单的webSocket测试工具。

简单的日志查看(TODO)。

web版的类文件生成(TODO)。

等等各种关于服务器和应用的信息查看。

//

(4)、更多功能

后续会根据用户需要增加更多的帮助工具，欢迎用户提供意见和贡献代码。

//

(5)、仓库

github https://github.com/swoft-cloud/swoft-devtool

//

(6)、参与贡献

您可以 fork 仓库修改然后发起 PR。

--

1、实体生成

实体生成,能快速生成实体文件,swoft 中提供了一套简单好用的生成方式.目前官方仅支持 mysql实体生成,其他类型的数据库参照官方例子.

//

(1)、安装

使用 composer 安装 composer require swoft/devtool

//

(2)、使用

php bin/swoft entity:gen

php bin/swoft entity:c

php bin/swoft entity:create [tableName] [--pool=xx] [--path=xx] [--table\_prefix=xx] [--field\_prefix=xxx] [--exc=xxx] [--td=xxx] [-y]

//

// 这三个命名是一个意思，可以使用 php bin/swoft entity:c -h 查看具体命令帮助文档

|1>、tableName 指定生成实体的数据表，如果为空会生成匹配到的所有表实体，也可以使用 --table=tableName 来指定数据表名，多个表用“,”隔开，如果设置了数据表前缀，tableName 不要带上表前缀。

|2>、pool 指定数据库连接池，默认会使用db.pool连接池。

|3>、path 生成实体的路径，支持 alias 方式传递路径，默认生成实体路径为@app/Model/Entity 比如: @app/Model/Entity中的@app会解析到项目的绝对路径。

|4>、table\_prefix 根据数据表前缀生成，简写是 tp，该参数不会受db中的参数限制，prefix底层会通过 like 进行表前缀匹配。

|5>、field\_prefix 去掉字段名前缀，简写是fp, 例如数据库字段名t\_name,生成实体的时候 加上--fp=t\_，生成之后的实体属性名为name。

|6>、exc 不期望生成的数据表名,使用场景例如你需要生成全部的实体部分表不希望生成实体就可以使用它，多个表用“,”隔开。

|7>、td 指定生成模板的路径，支持 alias方式传递路径，默认使用的模板路径是'@devtool/devtool/resource/template'。

|8>、y 生成文件是否需要确认, -y 参数加上就会直接生成文件不会确认提示。

//

(3)、特别注意

如果连接池中的db配置中存在prefix表前缀配置，生成的实体名会自动去掉表前缀，其中[tableName]、--table、[--exc=xxx]这两个参数查询数据表的时候自动会带上表前缀。请先检查 db的charset参数是否配置，不然生成实体中文注释将会乱码。

--

2、数据迁移 ( Swoft 的数据库迁移 Migrations )

(1)、简介

数据库迁移就像是数据库的版本控制，可以让你的团队轻松修改并共享应用程序的数据库结构。迁移通常会搭配上 Swoft 的数据库结构构造器来让你方便地构建数据库结构。如果你曾经出现过让同事手动在数据库结构中添加字段的情况，数据库迁移可以解决你这个问题。

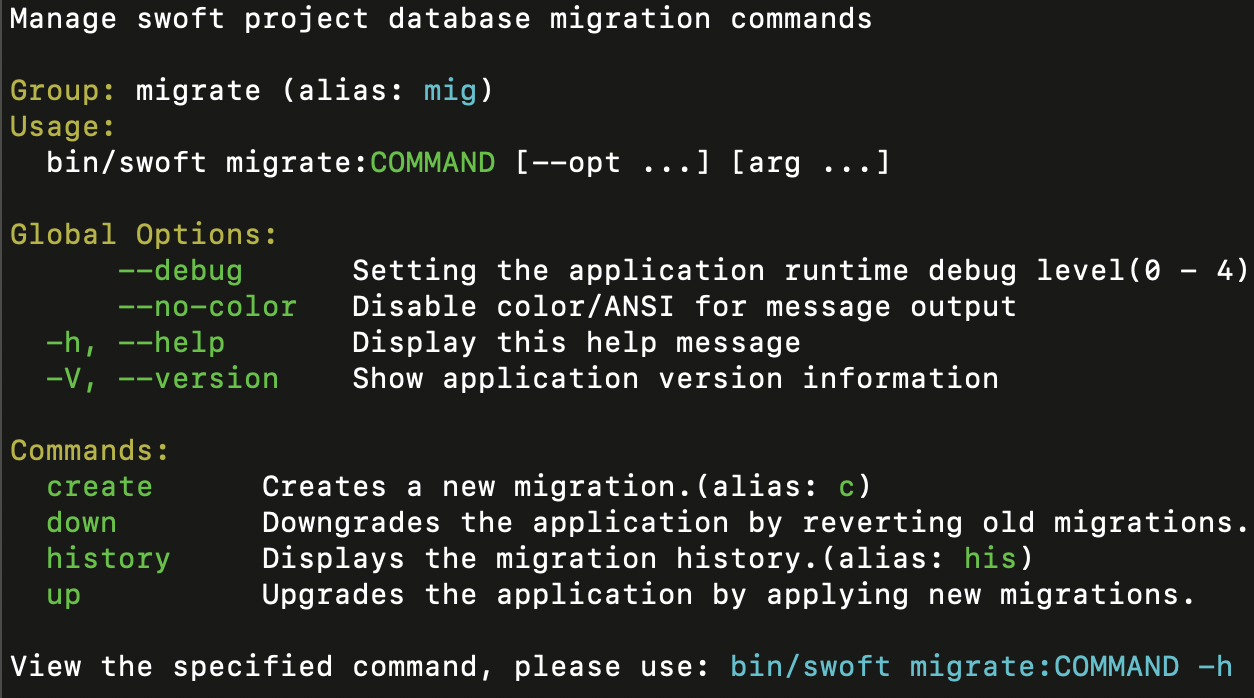
Swoft 的 Schema 对所有 Swoft 支持的数据库系统提供了创建和操作数据表的相应支持。

//

(2)、migrate 命令

php bin/swoft migrate

( 查看帮助文档图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrate.png )



//

//// create 迁移生成

// 使用 migrate:create 来创建迁移

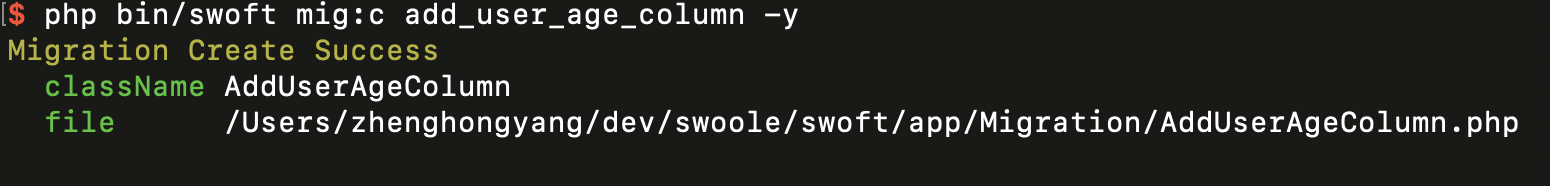
php bin/swoft migrate:create add\_user\_age\_column

// create 参数列表

name 指定迁移名称。

y 直接生成 不询问是否需要生成目录/文件。

( 你也可以使用 mig:c 来快速生成，如图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateCreate.png )



提示：同一目录迁移文件不可以重复生成。

新的迁移文件将默认会被放置在 App/Migration 目录中。每个迁移文件的名称不包含时间戳，在 @Migration默认参数就是，确认迁移的顺序。

//

// 如果你想迁移文件放置在自定义目录下，需要覆盖配置 migrationManager Bean 的 migrationPath 配置

// in bean.php file

'migrationManager' => [

    'migrationPath' => '@app/Migration',

],

//

(3)、迁移结构

一个迁移类会包含两个方法： up 和 down 。 up 方法可为数据库添加新的数据表、字段或索引，而 down 方法则是 up 方法的逆操作。

你可以在这两个方法中使用 Swoft 数据库结构构造器来创建以及修改数据表。若要了解 数据库结构 构造器中的所有可用方法，可查阅它的文档。以下的迁移实例会创建一张 Messages 数据表。

<?php declare(strict\_types=1);

namespace App\Migration;

use ReflectionException;

use Swoft\Bean\Exception\ContainerException;

use Swoft\Db\Exception\DbException;

use Swoft\Db\Schema\Blueprint;

use Swoft\Devtool\Annotation\Mapping\Migration;

use Swoft\Devtool\Migration\Migration as BaseMigration;

/\*\*

 \* Class Message

 \*

 \* @since 2.0

 \*

 \* @Migration(time=20190627225525, pool="db2.pool")

 \*/

class Message extends BaseMigration

{

    /\*\*

     \* @return void

     \*

     \* @throws ReflectionException

     \* @throws ContainerException

     \* @throws DbException

     \*/

    public function up(): void

    {

        $this->schema->createIfNotExists('messages', function (Blueprint $blueprint) {

            $blueprint->increments('id');

            $blueprint->text('content');

            $blueprint->timestamps();

        });

    }

    /\*\*

     \* @return void

     \*

     \* @throws ReflectionException

     \* @throws ContainerException

     \* @throws DbException

     \*/

    public function down(): void

    {

        $this->schema->dropIfExists('messages');

    }

}

?>

//

迁移必须要继承 Swoft\Devtool\Migration\Contract\MigrationInterface 契约，在基础抽象类 BaseMigration 中已经实现了 MigrationInterface 契约。在 BaseMigration 中提供了：schema 属性，该属性是一个 Schema Builder，提供了 和 $this->schema->xxx 一样的方法，底层会根据 @Migration 注解里面的 pool 参数选择好连接池注入到 schema 属性中；如果需要执行原生 SQL 可以使用 execute 方法, 在底层执行迁移的时候，这些 SQL 会被自动执行。

/\*\*

 \* @return void

 \*/

public function down(): void

{

    $truncate = <<<sql

truncate `users`;

sql;

    $this->execute($truncate);

    $dropSql = <<<sql

drop table if exists `users`;

sql;

    $this->execute($dropSql);

}

Swoft 迁移是每一个文件是一个事务，一个迁移文件中不宜写太多执行耗时长的 SQL, 以至于写锁等待时间太长, 如果执行中发生了异常会自动回滚。

//

//// @Migration 注解

// 迁移注解有两个参数

time 时间戳。

pool 当前迁移使用的连接池，默认为 db.pool。

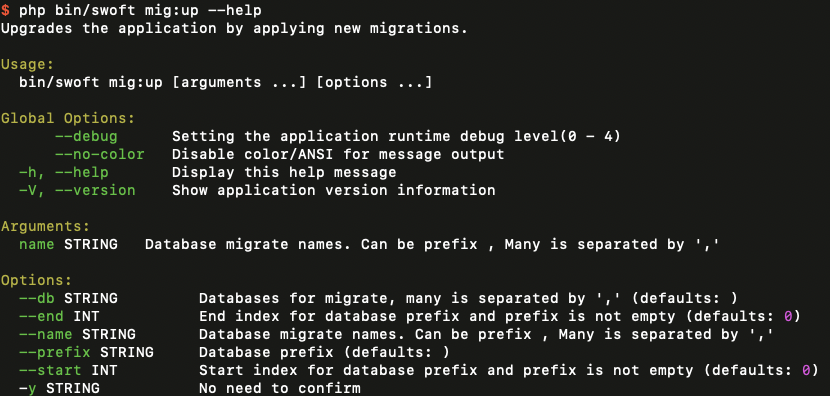
time 参数 是默认是你可以这样使用它 @Migration(20190630164222)，该参数必须存在 以便 Swoft 执行迁移。更多 demo 参考 Swoft 主项目下的 App\Migrations 目录。

//

(4)、运行迁移

///1/ 分库的一些支持

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateUp.png )



//

// 在运行 up, down, history 这三个命令的时候可以上如下参数

--db= 指定运行的 database，多个用逗号分隔。

--start= 开始的数据库 id。

--end= 结束的数据库 id。

--prefix= 指定执行 数据库运行的前缀, 这个参数只会影响 --db=, --start= , --end= 这三个参数的, 如果没有指定 --prefix= 会使用连接中的数据库名作为前缀。

//

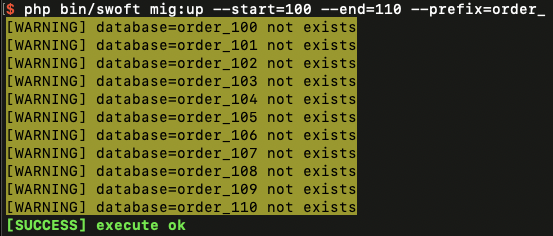
// 简单分库，假设后缀编号是自增的，例如

php bin/swoft migrate:up --prefix=order --start=1000 --end=1400

//

类似这样的操作会执行 order\_1000 到 order\_1400 之间的所有数据库, 如果有些不存在会自动过滤掉。

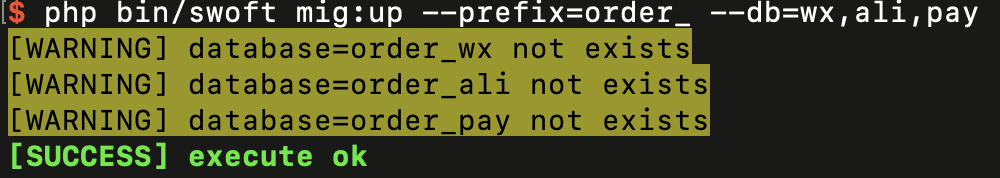
( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateUpPrefix.png )



//

// 还可以指定要执行的数据库，类似下面这个例子

( 图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateUpDb.png )



//

当然你也可以在迁移脚本的时候定义 @Migration 注解的 pool 参数执行的时候也会自动选择对应连接池的连接，这样也可以实现简单分库。

前提是你必须使用系统 BaseMigration 提供的 schema 属性、execute 方法，才会执行正确, 不然你需要手动自动连接池。

//

例如下面这样的, 这样不是很方便管理. 如果和定义 @Migration 注解的 pool 可能就会出问题, 如果需要指定连接池执行, 请使用 上面的 Demo 方式.

/\*\*

 \* @return void

 \*

 \* @throws ReflectionException

 \* @throws ContainerException

 \* @throws DbException

 \*/

public function down(): void

{

    $this->schema->dropIfExists('users');

    $this->schema->getSchemaBuilder('db2.pool')->dropIfExists('users');

}

//

///2/ 在线上环境强制执行迁移

一些迁移的操作是具有破坏性的，它们可能会导致数据丢失。为了保护线上环境的数据库，系统会在这些命令被运行之前显示确认提示。若要忽略此提示并强制运行命令，则可以使用 -y 标记。

//

// 强制执行

php bin/swoft migrate:up -y

//

///3/ 执行

// 使用 migrate:up 命令，来运行所有未运行过的迁移

php bin/swoft migrate:up

指定名称 执行, 这可能会匹配到多个 底层会 使用 stripos 匹配输入的字符在任意位置存在都可以不区分大小写, 下划线也是支持的底层会自动转换成驼峰进行匹配。

( 运行图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateAdd.png )



如果没有匹配到 将会中断执行。

//

///4/ 回滚迁移

// 若要回滚最后一次迁移，则可以使用 down 命令。此命令是对上一次最后一个执行的迁移回滚。

php bin/swoft migrate:down

//

// 在 down 命令后加上 step 参数，你可以限制回滚迁移的个数。例如，下面的命令将会回滚最后的 5 个迁移。

php bin/swoft migrate:down --step=5

//

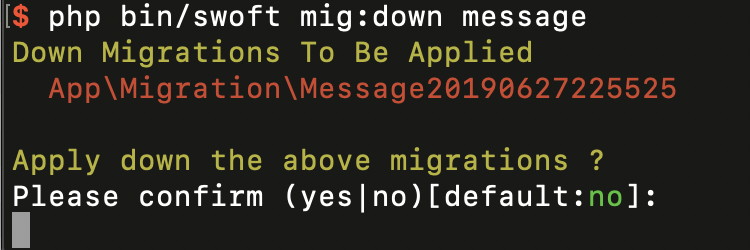
// 默认会从 db.pool 中获取连接 查询 迁移记录表, 也可以指定连接池。

php bin/swoft migrate:down --step=5 --pool=db.pool2

//

// 可指定迁移文件 回滚

( 原型图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateMessge.png )



//

///5/ 显示迁移历史

// 显示执行的 up/down 历史

php bin/swoft migrate:his

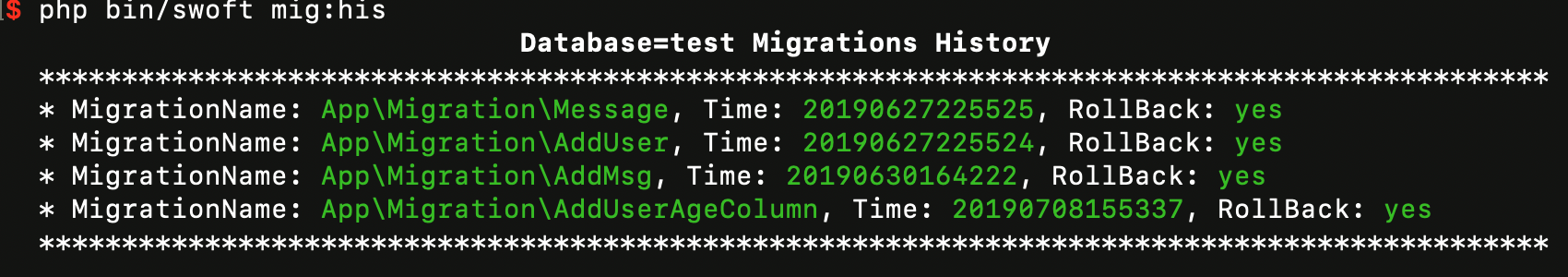
php bin/swoft migrate:history

//

// 默认会从 db.pool 中获取连接 查询 迁移记录表, 也可以指定连接池。

php bin/swoft mig:his --pool=db.pool2

( 运行图 https://raw.githubusercontent.com/swoft-cloud/swoft-doc/2.x/zh-CN/image/tool/migrateHis.png )



//

(5)、数据表

///1/ 创建数据表

要创建一张新的数据表，可以使用 Schema facade 的 create 方法。create 方法接收两个参数：第一个参数为数据表的名称，第二个参数为一个 闭包 ，此闭包会接收一个用于定义新数据表的 Blueprint 对象。

$this->schema->create('users', function (Blueprint $table) {

        $table->increments('id');

});

当然，在创建数据表的时候，你也可以使用任何数据库结构构造器的 字段方法( http://swoft.org/docs/2.x/zh-CN/tool/devtool/migrations.html#%E5%AD%97%E6%AE%B5 ) 来定义数据表的字段。

//

/// 检查数据表或字段是否存在

// 你可以方便地使用 hasTable 和 hasColumn 方法来检查数据表或字段是否存在。

if ($this->schema->hasTable('users')) {

    //

}

if ($this->schema->hasColumn('users', 'email')) {

    //

}

//

/// 数据库连接与存储引擎

// 如果你想要在一个非默认的数据库连接中进行数据库结构操作，可以使用 getSchemaBuilder 方法。

$this->schema->getSchemaBuilder('db.pool2')->create('users', function (Blueprint $table) {

    $table->increments('id');

});

//

/// 存储引擎：你可以在数据库结构构造器上设置 engine 属性来设置数据表的存储引擎, Mysql 默认为 InnoDB 。

$this->schema->create('users', function (Blueprint $table) {

    $table->engine = 'InnoDB';

    $table->increments('id');

});

//

/// 如果要判断数据表已存在就不创建可以使用 createIfNotExists 方法只针对可用的字段类型有效其他方法无法判断是否已经被创建过了。

$this->schema->createIfNotExists('users', function (Blueprint $blueprint) {

        $blueprint->increments('id');

});

//

///2/ 重命名与删除数据表

// 重命名：若要重命名一张已存在的数据表，可以使用 rename 方法。

$this->schema->rename($from, $to);

//

// 删除：要删除已存在的数据表，可使用 drop 或 dropIfExists 方法；

$this->schema->drop('users');

$this->schema->dropIfExists('users');

//

// 重命名带外键的数据表

在重命名前，你需要检查外键的约束涉及到的数据表名需要在迁移文件中显式的提供，而不是让 Swoft 按照约定来设置一个名称。因为那样会让外键约束关联到旧的数据表上。

//

(6)、字段

///1/ 创建字段

使用 Schema facade 的 table 方法可以更新已有的数据表。如同 create 方法，table 方法会接收两个参数：一个是数据表的名称，另一个则是接收 Blueprint 实例的闭包。我们可以使用它来为数据表新增字段。

$this->schema->table('users', function (Blueprint $table) {

        $table->string('email');

});

//

///2/ 添加表评论

使用 comment 方法 可以为数据表添加上注释。添加评论之前, 请先检查是否设置了 charset 不然生成的数据表注释，将会是乱码。

$this->schema->table('users', function (Blueprint $table) {

        $table->comment('this users table comment wow !');

});

//

// 可用的字段类型

数据库结构构造器包含了许多字段类型，供你构建数据表时使用。

(命令 描述)

$table->bigIncrements('id');  递增 ID（主键），相当于「UNSIGNED BIG INTEGER」型态。

$table->bigInteger('votes');  相当于 BIGINT 型态。

$table->binary('data'); 相当于 BLOB 型态。

$table->boolean('confirmed'); 相当于 BOOLEAN 型态。

$table->char('name', 4);  相当于 CHAR 型态，并带有长度。

$table->date('created\_at'); 相当于 DATE 型态

$table->dateTime('created\_at'); 相当于 DATETIME 型态。

$table->dateTimeTz('created\_at'); DATETIME (带时区) 形态

$table->decimal('amount', 5, 2);  相当于 DECIMAL 型态，并带有精度与基数。

$table->double('column', 15, 8);  相当于 DOUBLE 型态，总共有 15 位数，在小数点后面有 8 位数。

$table->enum('choices', ['foo', 'bar']);  相当于 ENUM 型态。

$table->float('amount', 8, 2);  相当于 FLOAT 型态，总共有 8 位数，在小数点后面有 2 位数。

$table->increments('id'); 递增的 ID (主键)，使用相当于「UNSIGNED INTEGER」的型态。

$table->integer('votes'); 相当于 INTEGER 型态。

$table->ipAddress('visitor'); 相当于 IP 地址形态。

$table->json('options');  相当于 JSON 型态。

$table->jsonb('options'); 相当于 JSONB 型态。

$table->longText('description');  相当于 LONGTEXT 型态。

$table->macAddress('device'); 相当于 MAC 地址形态。

$table->mediumIncrements('id'); 递增 ID (主键) ，相当于「UNSIGNED MEDIUM INTEGER」型态。

$table->mediumInteger('numbers'); 相当于 MEDIUMINT 型态。

$table->mediumText('description');  相当于 MEDIUMTEXT 型态。

$table->morphs('taggable'); 加入整数 taggable\_id 与字符串 taggable\_type。

$table->nullableMorphs('taggable'); 与 morphs() 字段相同，但允许为NULL。

$table->nullableTimestamps(); 与 timestamps() 相同，但允许为 NULL。

$table->rememberToken();  加入 remember\_token 并使用 VARCHAR(100) NULL。

$table->smallIncrements('id');  递增 ID (主键) ，相当于「UNSIGNED SMALL INTEGER」型态。

$table->smallInteger('votes');  相当于 SMALLINT 型态。

$table->softDeletes();  加入 deleted\_at 字段用于软删除操作。

$table->string('email');  相当于 VARCHAR 型态。

$table->string('name', 100);  相当于 VARCHAR 型态，并带有长度。

$table->text('description');  相当于 TEXT 型态。

$table->time('sunrise');  相当于 TIME 型态。

$table->timeTz('sunrise');  相当于 TIME (带时区) 形态。

$table->tinyInteger('numbers'); 相当于 TINYINT 型态。

$table->timestamp('added\_on');  相当于 TIMESTAMP 型态。

$table->timestampTz('added\_on');  相当于 TIMESTAMP (带时区) 形态。

$table->timestamps(); 加入 created\_at 和 updated\_at 字段。

$table->timestampsTz(); 加入 created\_at and updated\_at (带时区) 字段，并允许为NULL。

$table->unsignedBigInteger('votes');  相当于 Unsigned BIGINT 型态。

$table->unsignedInteger('votes'); 相当于 Unsigned INT 型态。

$table->unsignedMediumInteger('votes'); 相当于 Unsigned MEDIUMINT 型态。

$table->unsignedSmallInteger('votes');  相当于 Unsigned SMALLINT 型态。

$table->unsignedTinyInteger('votes'); 相当于 Unsigned TINYINT 型态。

$table->uuid('id'); 相当于 UUID 型态。

//

///3/ 字段修饰

除了上述的字段类型列表，还有一些其它的字段「修饰」，你可以将它增加到字段中。例如，若要让字段「nullable」，那么你可以使用 nullable 方法。

$this->schema->table('users', function (Blueprint $table) {

        $table->string('email')->nullable();

});

以下列表为字段的可用修饰。此列表不包括 索引修饰( http://swoft.org/docs/2.x/zh-CN/tool/devtool/migrations.html#%E7%B4%A2%E5%BC%95 )。

(Modifier Description)

->after('column') 将此字段放置在其它字段「之后」（仅限 MySQL）

->comment('my comment') 增加注释

->default($value) 为此字段指定「默认」值

->first() 将此字段放置在数据表的「首位」（仅限 MySQL）

->nullable()  此字段允许写入 NULL 值

->storedAs($expression) 创建一个存储的生成字段 （仅限 MySQL）

->unsigned()  设置 integer 字段为 UNSIGNED

->virtualAs($expression)  创建一个虚拟的生成字段 （仅限 MySQL）

//

///4/ 修改字段

/// 重命名字段/修改字段属性

要重命名字段，可使用数据库结构构造器的 renameColumn 方法。

$this->schema->table($table, function (Blueprint $blueprint) {

   // Rename column

   $blueprint->renameColumn('id', 'user\_id', 'bigint', 20);

});

//

///5/ 移除字段

要移除字段，可使用数据库结构构造器的 dropColumn 方法。

$this->schema->table('users', function (Blueprint $table) {

    $table->dropColumn('votes');

});

//

// 你可以传递多个字段的数组至 dropCloumn 方法来移除多个字段。

$this->schema->table('users', function (Blueprint $table) {

    $table->dropColumn(['votes', 'avatar', 'location']);

});

//

(7)、索引

///1/ 创建索引

数据库结构构造器支持多种类型的索引。首先，让我们先来看一个示例，其指定了字段的值必须是唯一的。你可以简单的在字段定义之后链式调用 unique 方法来创建索引。

$table->string('email')->unique();

//

// 可以在定义完字段之后创建索引。

$table->unique('email');

//

// 复合索引：可以传递一个字段的数组至索引方法来创建复合索引。

$table->index(['account\_id', 'created\_at']);

//

// 索引名称：Swoft 会自动生成一个合理的索引名称，但你也可以使用第二个参数来自定义索引名称。

$table->index('email', 'my\_index\_name');

//

///2/ 可用的索引类型

(Command Description)

$table->primary('id');  加入主键。

$table->primary(['first', 'last']); 加入复合键。

$table->unique('email');  加入唯一索引。

$table->unique('state', 'my\_index\_name'); 自定义索引名称。

$table->unique(['first', 'last']);  加入复合唯一键。

$table->index('state'); 加入基本索引。

//

///3/ 索引长度 & MySQL / MariaDB

Swoft 默认使用 utf8mb4 字符，包括支持在数据库存储「表情」。如果你正在运行的 MySQL release 版本低于5.7.7 或 MariaDB release 版本低于10.2.2 ，为了MySQL为它们创建索引，你可能需要手动配置迁移生成的默认字符串长度， $this->schema->defaultStringLength 方法来配置它。

$this->schema->defaultStringLength(191);

或者你可以为数据库开启 innodb\_large\_prefix 选项，有关如何正确开启此选项的说明请查阅数据库文档。

//

///4/ 移除索引

若要移除索引，则必须指定索引的名称。Swoft 默认会自动给索引分配合理的名称。其将数据表名称、索引的字段名称及索引类型简单地连接在了一起。举例如下所示。

(命令 描述)

$table->dropPrimary('users\_id\_primary');  从「users」数据表移除主键。

$table->dropUnique('users\_email\_unique'); 从「users」数据表移除唯一索引。

$table->dropIndex('geo\_state\_index'); 从「geo」数据表移除基本索引。

//

如果你对 dropIndex 传参索引数组，默认的约定是索引名称由数据库表名字和键名拼接而成。

$this->schema->table('geo', function (Blueprint $table) {

    $table->dropIndex(['state']); // 移除索引 'geo\_state\_index'

});

//

(8)、外键约束

Swoft 也为创建外键约束提供了支持，用于在数据库层中的强制引用完整性。例如，让我们定义一个有 user\_id 字段的 posts 数据表，user\_id 引用了 users 数据表的 id 字段。

$this->schema->table('posts', function (Blueprint $table) {

        $table->integer('user\_id')->unsigned();

        $table->foreign('user\_id')->references('id')->on('users');

});

//

// 可以指定约束的「on delete」及「on update」

$table->foreign('user\_id')

  ->references('id')->on('users')

  ->onDelete('cascade');

//

要移除外键，你可以使用 dropForeign 方法。外键约束与索引采用相同的命名方式。所以，我们可以将数据表名称和约束字段连接起来，接着在该名称后面加上「\_foreign」后缀。

$table->dropForeign('posts\_user\_id\_foreign');

//

你也可以传递一个包含字段的数组，在移除的时候字段会按照惯例被自动转换为对应的外键名称。

$table->dropForeign(['user\_id']);

//

// 你可以在迁移文件里使用以下方法来开启和关闭外键约束。

$this->schema->enableForeignKeyConstraints();

$this->schema->disableForeignKeyConstraints();

--

--

第七篇 扩展