

轻松使用

SwooleDistributed3.X



目 录

SD3.X简介

捐赠SD项目

VIP服务

基础篇

搭建环境

启动命令

开发注意事项

框架配置

配置文件夹

server.php

ports.php

business.php

mysql.php

redis.php

timerTask.php

log.php

consul.php

catCache.php

client.php

自定义配置

框架入口

MVC架构

加载器-Loader

控制器-Controller

模型-Model

视图-View

同步任务-Task

封装器

Swoole编程指南-EOF协议

Swoole编程指南-固定包头协议

封装器-Pack

路由器

TCP相关

绑定UID

Send系列

Sub/Pub

获取服务器信息

Http相关

HttpInput

- HttpOutput
- 默认路由规则
- WebSocket相关
- 使用SSL
- 公共函数
- 进阶篇
 - 内核优化
 - 对象池
 - 上下文-Context
 - 中间件
 - 进程管理
 - 创建自定义进程
 - 进程间RPC
 - 自定义进程如何使用连接池
 - 异步连接池
 - Redis
 - Mysql
 - Mqtt
 - HttpClient
 - Client
 - AMQP
 - RPC
 - 日志工具-GrayLog
 - 微服务-Consul
 - Consul基础
 - 搭建Consul服务器
 - SD中Consul配置
 - 微服务
 - 选举-Leader
 - Consul动态配置定时任务
 - 熔断与降级
 - 集群-Cluster
 - 高速缓存-CatCache
 - 万物-Actor
 - Actor原型
 - Actor的创建
 - Actor间的通讯
 - 消息派发-EventDispatcher
 - 例子A
 - 延迟队列-TimerCallBack
 - 协程
 - 订阅与发布

MQTT简易服务器

AMQP异步任务调度

自定义命令-Console

调试工具Channel

特别注意事项

日常问题总结

实践案例

物联网自定义协议

Actor在游戏的应用

Mongodb以及一些同步扩展的使用

开发者工具

SD3.X简介

SD3.X简介

SD3.X版本是SD2.X的升级版，拥有SD2.X完整功能，并在2.X的基础上进行了大量的优化，其主要特性在与3.X版本使用了swoole2.x版本的扩展，利用swoole2.x内置的协程机制取代了php yield，使得书写代码上更加简洁，如同同步书写一样，可以轻松的使用。3.X兼容2.X的绝大部分代码，2.X升级到3.X相对来说比较轻松，会有专门章节介绍迁移事项。

优势

- SD环境一键安装
- SD项目成立时间长，经历了多个线上项目的迭代，稳定性优异。
- 拥有众多的功能模块，帮助你实现业务代码的编写
- 天然支持分布式部署
- 拥有微服务架构，RPC支持
- 支持多端口多协议，可同时支持UDP，TCP，HTTP，HTTPS，WS，WSS等协议
- 异步连接库丰富，支持redis，mysql，httpclient，tcpclient，mqttclient
- 进程管理完善，自定义进程管理和进程间通讯支持完善
- 支持定时任务可配置
- 支持AMQP分布式任务调度
- MVC设计模式
- 对象池内存管理，减少GC
- cli命令支持，可自定义命令
- VIP服务，支持1对1教学

全自动安装部署

SD提供提供了运行环境的一键安装部署，可用于全新系统的环境搭建。

执行下面命令将安装SD3.X版本，包括所有环境，省心省事。

```
curl -sS sd.youwoxing.net/install_v3 | bash
```

将安装部署的有：PHP7.1.14,Swoole2.X,以及框架需要的各种扩展。

智能路由

通过编写自定义封装器将获得的消息解包。

通过编写自定义路由可以将消息投递到不同的控制器中。

通过调用Model可以将业务解耦。

多协议支持

可以开启多个端口同时支持多种协议。

支持的协议有UDP，TCP，WS，HTTP，WSS，HTTPS，满足日常开发需要。

不同端口可以拥有不同的封装器，框架会根据封装器自动封装消息下发到客户端，无需手动操作。

进程管理

用户可以根据具体业务创建自定义进程，框架封装好了进程的管理，可以轻松创建自定义进程。

自定义进程和worker进程间可以互相通讯，自定义进程间也可以互相通讯。

Mysql/Redis连接池/协程

框架封装了Mysql和Redis连接池，用户调用方法时和同步用法一致，却是异步的性能。

框架对协程的使用也进行了封装，高级用户可以通过继承协程基类制作自己的异步协程方法。

发布/订阅

框架提供发布订阅接口，严格满足MQTT协议定制的标准，却不需要MQTT协议支持，可以在任何协议中使用。

MQTT

提供MQTT异步客户端，可以接入开源MQTT服务器。

框架也可以开启简易MQTT服务器，实现MQTT基础功能。

AMQP分布式任务调度系统

框架提供AMQP任务调度支持，可以通过RabbitMQ实现AMQP异步任务调度。

通过开启自定义进程，可以为框架增加任务调度系统。

Event

框架提供了Event事件模块，可以通过监听事件实现异步调用。并且这些事件可以在集群内广播。

GrayLog日志系统

接入了GrayLog日志系统，可以通过搭建GrayLog实现一套大型日志处理系统。

定时任务

通过TimerTask配置实现定时任务。

集群

框架设计本身自带集群基因，所有的消息API均支持集群。

微服务

通过Consul实现服务注册和监控，框架提供RPC方案。

Actor

Actor模型，用于游戏开发，Actor间可以互相通讯，支持事务，支持分布式部署。
Actor很轻，服务器可以开启几十万个，用于对游戏内单位进行模拟。

Reload

开启自动Reload后，代码发生改变无需重启服务器。

CatCache

可落地的高速缓存，由PHP实现

捐赠SD项目

捐赠SD项目

- [捐赠SD项目](#)
- [捐赠方式](#)
 - [微信](#)
 - [支付宝](#)
- [捐赠列表](#)

您的捐赠是对SwooleDistributed项目最大的鼓励和支持。我们会坚持开发维护下去。

您的捐赠将被用于：

- 持续和深入地开发
- 文档和社区的建设与维护

捐赠方式

微信



微信捐赠请写备注留言捐赠人信息。

支付宝



捐赠列表

如果您捐赠了SwooleDistributed开源项目但不在下面的列表中，或者希望修改相关信息，请联系白猫（QQ：896369042）。

日期	捐赠金额	捐赠者	寄语	渠道
2017-04-07 14:49	666.00	小杨 bad***@163.com		支付宝
2017-04-07 15:11	6.66	生活不能自理 156_**_6173		支付宝
2017-04-07 15:15	88.00	子斌 son***@live.com		支付宝
2017-05-10 13:45	30.00	大眼猫 xwm_*_@126.com	thankyou	支付宝
2017-07-13 16:02	100.88	恹心	感谢开发了个那么棒的项目，段云飞留	支付宝
2017-07-13 16:02	400.00	恹心	感谢	支付宝
2017-07-13 16:02	400.00	too	感谢	支付宝
2017-07-	400.00	nemo	感谢	支付

				宝
2017-07-19 17:15	400.00	海涛	感谢	支付宝
2017-07-19 17:52	400.00		感谢	微信
2017-07-19 17:55	400.00	滔滔	感谢	支付宝
2017-07-31 22:07	400.00	瑞海	感谢	支付宝
2017-07-31 23:08	400.00	雪辉	感谢	支付宝
2017-08-08 15:53	400.00	纯情小姐	感谢	支付宝
2017-08-16 16:22	400.00	自豪	感谢	支付宝
2017-08-16 16:38	400.00	俊	感谢	支付宝
2017-08-16 16:53	400.00	洋洋	感谢	支付宝
2017-08-17 09:28	400.00	李二蛋	感谢	微信

17 09:28	400.00	李二蛋	感谢	信
2017-08-22 11:21	20.00	杨泽兴	支持SD	支付宝
2017-08-26 12:01	60.00	峰	感谢	支付宝
2017-08-31 14:43	10.80		感谢	微信
2017-09-03 14:27	400.00	锦泉	感谢	支付宝
2017-09-08 10:14	400.00	习广	感谢	支付宝
2017-09-08 11:21	400.00	军剑	感谢	支付宝
2017-09-08 11:21	1900.00	daydaygo (陈志林)	支持猫神, 支持 SD http://wiki.daydaygo.top	支付宝
2017-09-12 16:29	400.00	剑锋	感谢	支付宝
2017-09-20 18:55	404.00	东荣	感谢	支付宝
2017-09-	400.00	未知	感谢	微

2017-09-22 11:46	400.00	未知	感谢	微信
2017-09-26 18:51	400.00	荣祥	感谢	支付宝
2017-10-12 17:39	400.00	文文	感谢	支付宝
2017-10-16 10:22	400.00	杰	感谢	支付宝
2017-10-16 10:40	500.00	自豪	感谢	支付宝
2017-10-25 15:39	1.00	善利	感谢	支付宝
2017-10-26 10:09	400.00	建杰	感谢	支付宝
2017-10-30 13:41	400.00	伟	感谢	支付宝
2017-10-30 15:33	400.00	立派	感谢	支付宝
2017-10-30 16:13	400.00	未知	感谢	微信
				支

2017-10-31 16:20	400.00	则仲	感谢	支付宝
2017-10-31 21:52	400.00	志仰	感谢	支付宝
2017-11-02 20:30	1000.00	茂甘	感谢	支付宝
2017-11-16 16:48	1000.00	红飞	感谢	支付宝
2017-11-17 11:20	1000.00	来君	感谢	支付宝
2017-11-17 11:40	1000.00	小杨	感谢	微信
2017-11-22 10:23	6.66	乃文	感谢	支付宝

VIP服务

VIP服务

为了更好的持续性开源，以及针对个人企业提供更加优质的服务，SD框架提供VIP服务。

如何成为VIP

联系白猫QQ：896369042

VIP分批次招募，每批次价格有可能会有所调整，每批招募10人。

VIP的服务

- 小群模式及时反馈，技术氛围活跃
- 群内组员大多数都有SD在线项目的经验
- 提出功能性的需求可以优先被满足
- 加快你对框架的了解，避免踩坑
- 如遇到难点问题可以请求远程协助
- 免费提供SD框架运维后台，以后会有更多工具
- 一个上线的SD项目实例，以后会有更多例子
- 分布式，微服务，搭建讲解
- 各种高级功能的使用讲解

适合人群

- 需要使用SD搭建线上项目
- 希望深入了解SD框架
- 需要获得更优质的技术支持
- 需要使用更高级的功能
- 需要拓展自己的技术涉猎面

通过学习SD框架，你会了解更多的工具，更多的知识，更多的架构师思维。

基础篇

基础篇

搭建环境

搭建环境

自动安装

SD提供提供了运行环境的一键安装部署，可用于全新系统的环境搭建。

环境包括PHP，各类扩展，SD框架。

执行下面命令将安装SD3.X版本，包括所有环境

```
curl -sS sd.youwoxing.net/install_v3 | bash
```

运行后，喝杯茶静等安装完成。

如果你已经安装了PHP，有可能会出现安装错误，请移除你安装的PHP版本再执行上面的命令。

该命令已在ubuntu，centos，deepin系统中验证有效。

上面的命令会自动生成SD的项目文件夹，可以修改composer.json文件增加自己的依赖或者修改SD的版本。

composer.json：

```
{
  "require": {
    "tmtbe/swooledistributed": ">3.0.0"
  },
  "autoload": {
    "psr-4": {
      "app\\": "src/app",
      "test\\": "src/test"
    }
  }
}
```

通过composer进行依赖代码更新，运行：

```
composer update
```

手动安装

1. 安装PHP7.1.14版本，包含bcmath，mbstring，pdo_mysql，posix扩展。

2. 安装swoole , inotify , ds , redis , hiredis

使用下面的安装脚本安装，可以通过修改版本号安装不同版本。

```
#!/bin/sh
#sd安装脚本
set -e
pwd=`pwd`
hiredis_version="0.13.3"
phpredis_version="3.1.6"
phpds_version="1.2.4"
phpinotify_version="2.0.0"
swoole_version="2.1.1"
sd_version="3.0.6"
swoole_configure="--enable-async-redis --enable-openssl --enable-coroutine"

do_install_ex() {
    php_m=`php -m`
    if ! [[ $php_m =~ "bcmath" ]] ; then
        echo "缺少必要bcmath扩展，请安装或者卸载php版本后重新运行本命令"
        exit
    fi
    if ! [[ $php_m =~ "mbstring" ]] ; then
        echo "缺少必要mbstring扩展，请安装或者卸载php版本后重新运行本命令"
        exit
    fi
    if ! [[ $php_m =~ "pdo_mysql" ]] ; then
        echo "缺少必要pdo_mysql扩展，请安装或者卸载php版本后重新运行本命令"
        exit
    fi
    if ! [[ $php_m =~ "posix" ]] ; then
        echo "缺少必要posix扩展，请安装或者卸载php版本后重新运行本命令"
        exit
    fi

    do_install_swoole

    if ! [[ $php_m =~ "redis" ]] ; then
        do_install_redis
    fi
    if ! [[ $php_m =~ "ds" ]] ; then
        do_install_ds
    fi
    if ! [[ $php_m =~ "inotify" ]] ; then
        do_install_inotify
    fi
}

do_install_inotify() {
    cd ${pwd};
    echo "[inotify] 开始下载"
```

```

if [ -f "${phpinotify_version}.tar.gz" ] ;then
    rm -f ${phpinotify_version}.tar.gz
fi
wget https://github.com/arnaud-lb/php-inotify/archive/${phpinotify_version}.tar.g
z
if [ -d "php-inotify-${phpinotify_version}" ] ;then
    rm -rf php-inotify-${phpinotify_version}
fi
tar -xzvf ${phpinotify_version}.tar.gz > /dev/null
echo "[inotify] 开始编译"
cd ${pwd}/php-inotify-${phpinotify_version}
phpize
./configure
make clean > /dev/null
make -j
make install
echo "extension=inotify.so" | tee ${init_file}/inotify.ini
rm -f ${pwd}/${phpinotify_version}.tar.gz
rm -rf ${pwd}/php-inotify-${phpinotify_version}
}

do_install_ds() {
    cd ${pwd};
    echo "[ds] 开始下载"
    if [ -f "v${phpds_version}.tar.gz" ] ;then
        rm -f v${phpds_version}.tar.gz
    fi
    wget https://github.com/php-ds/extension/archive/v${phpds_version}.tar.gz
    if [ -d "extension-${phpredis_version}" ] ;then
        rm -rf extension-${phpredis_version}
    fi
    tar -xzvf v${phpds_version}.tar.gz > /dev/null
    echo "[ds] 开始编译"
    cd ${pwd}/extension-${phpds_version}
    phpize
    ./configure
    make clean > /dev/null
    make -j
    make install
    echo "extension=ds.so" | tee ${init_file}/ds.ini
    rm -f ${pwd}/v${phpds_version}.tar.gz
    rm -rf ${pwd}/extension-${phpredis_version}
}

do_install_redis() {
    cd ${pwd};
    echo "[redis] 开始下载"
    if [ -f "${phpredis_version}.tar.gz" ] ;then
        rm -f ${phpredis_version}.tar.gz
    fi
    wget https://github.com/phpredis/phpredis/archive/${phpredis_version}.tar.gz
    if [ -d "redis-${phpredis_version}" ] ;then

```

```

        rm -rf redis-${phpredis_version}
    fi
    tar -xzf ${phpredis_version}.tar.gz > /dev/null
    echo "[redis] 开始编译"
    cd ${pwd}/${phpredis}-${phpredis_version}
    phpize
    ./configure
    make clean > /dev/null
    make -j
    make install
    echo "extension=redis.so" | tee ${init_file}/redis.ini
    rm -f ${pwd}/${phpredis_version}.tar.gz
    rm -rf ${pwd}/redis-${phpredis_version}
}

do_install_swoole() {
    cd ${pwd};
    echo "[swoole] 开始下载"
    if [ -f "v${swoole_version}.tar.gz" ] ;then
        rm -f v${swoole_version}.tar.gz
    fi
    if [ -f "v${hiredis_version}.tar.gz" ] ;then
        rm -f v${hiredis_version}.tar.gz
    fi
    wget https://github.com/swoole/swoole-src/archive/v${swoole_version}.tar.gz https
://github.com/redis/hiredis/archive/v${hiredis_version}.tar.gz
    if [ -d "swoole-src-${swoole_version}" ] ;then
        rm -rf swoole-src-${swoole_version}
    fi
    if [ -d "hiredis-${hiredis_version}" ] ;then
        rm -rf hiredis-${hiredis_version}
    fi
    tar -xzf v${swoole_version}.tar.gz > /dev/null
    tar -xzf v${hiredis_version}.tar.gz > /dev/null
    echo "[swoole] 开始编译"
    cd ${pwd}/hiredis-${hiredis_version}
    make clean > /dev/null
    make -j
    make install
    ldconfig
    cd ${pwd}/swoole-src-${swoole_version}
    phpize
    ./configure ${swoole_configure}
    make clean > /dev/null
    make -j
    make install
    echo "extension=swoole.so" | tee ${init_file}/swoole.ini
    rm -f ${pwd}/v${swoole_version}.tar.gz
    rm -f ${pwd}/v${hiredis_version}.tar.gz
    rm -rf ${pwd}/swoole-src-${swoole_version}
    rm -rf ${pwd}/hiredis-${hiredis_version}
}

```

```

do_install_sd(){
    echo "[SD] 开始安装"
    cd ${pwd}
    mkdir sd
    cd sd
    echo "{\"require\": {\"tmtbe/swooledistributed\": \"=${sd_version}\", \"autoload\"
: {\"psr-4\": {\"app\\\\\\\\\": \"src/app\\\\\", \"test\\\\\\\\\": \"src/test\\\\\"}}}\" > composer.js
on
    composer update
    php vendor/tmtbe/swooledistributed/src/Install.php -y
    echo "框架安装完毕,项目目录:${pwd}/sd"
}

do_install_composer(){
    if ! command_exists composer ; then
        cd ${pwd}
        echo "[composer] 开始下载"
        curl -sS https://getcomposer.org/installer | php
        mv composer.phar /usr/local/bin/composer
    fi
    composer config -g repo.packagist composer https://packagist.phpcomposer.com
}

set_php() {
    init_file=`echo '<?php $command=$argv[1]??null;ob_start();phpinfo(INFO_GENERAL);$
result=ob_get_contents();ob_clean();$ini_files="";$info=explode("\n\n",$result)[1];$
info2=explode("\n",$info);foreach($info2 as $value){$info3=explode("=>",$value);if($
info3[0]=="Scan this dir for additional .ini files "){$ini_files = trim($info3[1]);b
reak;}}echo $ini_files;' | php`
}

set_ldconf() {
    echo "include /etc/ld.so.conf.d/*.conf" | tee /etc/ld.so.conf
    if ! [ -d "/etc/ld.so.conf.d" ] ;then
        mkdir /etc/ld.so.conf.d
    fi
    cd /etc/ld.so.conf.d
    echo "/usr/local/lib" | tee /etc/ld.so.conf.d/libc.conf
    ldconfig -v
}

do_install(){
    set_ldconf
    php_version=`php -v`
    if ! [[ $php_version =~ "PHP 7" ]] ; then
        echo "PHP版本不对,请卸载php后重新运行此命令"
    fi
    set_php
    do_install_ex
    do_install_composer
    do_install_sd
}

```

```
}  
  
do_install
```

启动命令

启动命令

- 启动命令
 - 启动
 - 调试
 - 重启
 - 重载
 - 停止
 - 强杀
 - 单元测试

进入项目的bin目录下，可以执行list查看命令

```
php start_swoole_server.php list
Console Tool
```

Usage:

```
command [options] [arguments]
```

Options:

```
-h, --help           Display this help message
-q, --quiet          Do not output any message
-V, --version        Display this application version
    --ansi           Force ANSI output
    --no-ansi        Disable ANSI output
-n, --no-interaction Do not ask any interactive question
-v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 f
or more verbose output and 3 for debug
```

Available commands:

```
clear  Clear server actor and timer callback
help   Displays help for a command
kill   Kill server
list   Lists commands
reload Reload server
restart Restart server
start  Start server
status Server Status
stop   Stop(Kill) server
test   Test case
```

启动

调试模式启动服务器

```
php start_swoole_server.php start
```

守护进程启动服务器

```
php start_swoole_server.php start -d
```

调试

```
php start_swoole_server.php start -de
```

还可以附加过滤器

```
php start_swoole_server.php start -de --f abc
```

重启

会自动结束进程然后重新启动一个守护进程模式的服务器

```
php start_swoole_server.php restart
```

重载

不会断开客户端链接，进行代码的重载。升级服务器逻辑，客户端无感知。

```
php start_swoole_server.php reload
```

停止

停止服务器

```
php start_swoole_server.php stop
```

强杀

有时候会出现stop失败的情况，这时候可以使用kill命令强杀。


```
php start_swoole_server.php kill
```

单元测试

测试test目录下所有的测试类

```
php start_swoole_server.php test
```

测试test目录下指定的测试类

```
php start_swoole_server.php test XXXX
```

开发注意事项

开发注意事项

目前推荐PHP7.1.14作为开发环境

call_user_func_array和call_user_func的问题

call_user_func和call_user_func_array后调用的所有方法里都不能出现协程的操作，否则会引发崩溃现象。

可以使用\co::call_user_func_array和\co::call_user_func代替。

请求周期差异

PHP在Web应用中一次请求过后会释放所有的变量与资源

SD开发的应用程序在第一次载入解析后便常驻内存，使得类的定义、全局对象、类的静态成员不会释放，便于后续重复利用

注意Task的单例模式

通过loader获取的task其实是个TaskProxy，TaskProxy为单例模式请使用时再获取。

注意对象池模式

SD框架中大量使用了对象池模式，减少GC时间，缓解内存波动，提高运行效率。

注意不要使用sleep、exit、die

SD运行在PHP命令行模式下，当调用exit、die退出语句时，会导致当前进程退出。虽然子进程退出后会立刻重新创建一个的相同的子进程继续服务，但是还是可能对业务产生影响。

sleep命令会导致异步进程的堵塞，请使用sleepCoroutine代替

平滑重启

使用reload命令会进行平滑重启，需要注意的是app文件夹中的所有文件基本都能进行平滑重启的升级

框架配置

框架配置

框架的配置文件在config目录下，下面详细介绍各个文件的含义

配置文件夹

[server.php](#)

[ports.php](#)

[business.php](#)

[mysql.php](#)

[redis.php](#)

[timerTask.php](#)

[log.php](#)

[consul.php](#)

[catCache.php](#)

配置文件夹

配置文件夹

- [配置文件夹](#)
 - [配置文件位置](#)
 - [多配置目录](#)

配置文件位置

默认在SD根目录下存在config文件夹，里面内置了很多配置文件。

多配置目录

默认config文件夹下的php配置文件将会生效，但可以通过修改环境变量的方式可以选择config文件夹下的子文件夹作为配置生效文件夹。

SD框架提供了环境变量用于修改配置文件的生效目录

SD_CONFIG_DIR

例如SD_CONFIG_DIR=docker将设置config/docker目录为生效目录，可以把config下的配置文件拷贝至docker文件夹下。

server.php

server.php

服务器基础配置

```
/**
 * 服务器设置
 */
$config['server']['send_use_task_num'] = 20;
$config['server']['set'] = [
    'log_file' => LOG_DIR."/swoole.log",
    'log_level' => 5,
    'reactor_num' => 2, //reactor thread num
    'worker_num' => 4,    //worker process num
    'backlog' => 128,    //listen backlog
    'open_tcp_nodelay' => 1,
    'dispatch_mode' => 5,
    'task_worker_num' => 5,
    'task_max_request' => 5000,
    'enable_reuse_port' => true,
    'heartbeat_idle_time' => 120, //2分钟后没消息自动释放连接
    'heartbeat_check_interval' => 60, //1分钟检测一次
    'max_connection' => 100000
];

//协程超时时间
$config['coroutine']['timerOut'] = 5000;

//是否启用自动reload
$config['auto_reload_enable'] = true;

//是否允许访问Server中的Controller, 如果不允许将禁止调用Server包中的Controller
$config['allow_ServerController'] = true;
```

worker_num

开启的worker进程数量，这里建议设置为cpu核数一致，比如你是4核那么worker_num设置为4

task_worker_num

开启的task进程数量，task是同步进程主要处理耗时任务，这里如果用到task请设置task进程数。

Task是同步模式很多功能受到限制，SD3.0建议使用自定义进程实现耗时任务的执行。

heartbeat_idle_time

心跳，这里服务器将检测客户端是否有消息到达，如果消息间隔超过heartbeat_idle_time设置的值，那么该客户端将被踢下线，heartbeat_idle_time单位为秒

如果不想检测心跳可以删除此配置

heartbeat_check_interval

服务器间隔heartbeat_check_interval（单位秒）时间检测一次心跳

如果不想检测心跳可以删除此配置

max_connection

这里set中的max_connection是设置服务器最大连接数，如果超过则会拒绝。

max_connection越大申请的内存越大，运行时可能会报错，那么需要调整下面的参数。

ulimit -n 要调整为100000甚至更大。命令行下执行 ulimit -n 100000即可修改。如果不能修改，需要设置 /etc/security/limits.conf，加入

```
* soft nfile 262140
* hard nfile 262140
root soft nfile 262140
root hard nfile 262140
* soft core unlimited
* hard core unlimited
root soft core unlimited
root hard core unlimited
```

注意，修改limits.conf文件后，需要重启系统生效。

config['coroution']['timerOut']

设置默认的协程超时时间，这里单位为毫秒。

config['auto_reload_enable']

是否启动自动reload功能，开启后有代码更新会自动reload，不需要开发者重启服务器。

如果没有装inotify扩展会导致cpu占用率极高，建议安装inotify扩展
线上环境不建议开启

allow_ServerController

是否允许访问Server中的Controller，如果不允许将禁止调用Server包中的Controller

ports.php

ports.php

配置服务器端口信息，这是一个非常重要的配置

```
use Server\CoreBase\PortManager;

$config['ports'][] = [
    'socket_type' => PortManager::SOCK_TCP,
    'socket_name' => '0.0.0.0',
    'socket_port' => 9091,
    'pack_tool' => 'LenJsonPack',
    'route_tool' => 'NormalRoute',
];

$config['ports'][] = [
    'socket_type' => PortManager::SOCK_TCP,
    'socket_name' => '0.0.0.0',
    'socket_port' => 9092,
    'pack_tool' => 'EofJsonPack',
    'route_tool' => 'NormalRoute',
];

$config['ports'][] = [
    'socket_type' => PortManager::SOCK_HTTP,
    'socket_name' => '0.0.0.0',
    'socket_port' => 8081,
    'route_tool' => 'NormalRoute',
    'method_prefix' => 'http_'
];

$config['ports'][] = [
    'socket_type' => PortManager::SOCK_WS,
    'socket_name' => '0.0.0.0',
    'socket_port' => 8083,
    'route_tool' => 'NormalRoute',
    'pack_tool' => 'NonJsonPack',
    'opcode' => PortManager::WEBSOCKET_OPCODE_TEXT
];

return $config;
```

通过此设置可以配置开启多个端口，每个端口可以配置不同的封装器和路由器。

其中http不需要封装器可以不填写。

可以通过命名回调函数来为每个端口提供不同的回调。

可以配置的字段名为：


```

-----HTTP-----
request
handshake
-----WS-----
open
message
close
handshake
-----TCP/UDP-----
connect
receive
close
packet

```

还有些特殊的配置

- `method_prefix` 设置该端口访问的方法名前缀
- `event_controller_name` 设置该端口connect,close触发的控制器名称，不填默认使用Appserver设置的
- `close_method_name` 设置该端口close触发的方法，不填默认使用Appserver设置的
- `connect_method_name` 设置该端口connect触发的方法，不填默认使用Appserver设置的

如果仅仅使用http协议可以直接使用下面的配置

```

$config['ports'][] = [
    'socket_type' => PortManager::SOCK_HTTP,
    'socket_name' => '0.0.0.0', //0.0.0.0表示运行所有的ip访问
    'socket_port' => 8081, //开启的端口为8081
    'route_tool' => 'NormalRoute', //框架默认的路由就是NormalRoute
    'method_prefix' => 'http_' //控制器访问的前缀为'http_'
];

```

关于前缀

框架为了隔离访问添加了访问方法前缀

比如8081端口设置了HTTP服务，`method_prefix`设置了前缀为`http_`。

通过NormalRoute作为默认路由的情况下，我们访问`localhost:8081/Testcontroller/test`将访问到TestController控制器的`http_test`方法。

business.php

business.php

业务上的一些配置。

```
<?php
/**
 * Created by PhpStorm.
 * User: zhangjincheng
 * Date: 16-7-14
 * Time: 下午1:58
 */

//强制关闭gzip
$config['http']['gzip_off'] = false;

//默认访问的页面
$config['http']['index'] = 'index.html';

/**
 * 设置域名和Root之间的映射关系
 */

$config['http']['root'] = [
    'default' =>
        [
            'index' => ['TestController', 'test'] //转到控制器
        ]
    ,
    'localhost' =>
        [
            'root' => 'www',
            'index' => 'Index.html' //转到指定页面
        ]
];

return $config;
```

- 设置域名访问的时候对应的根目录名称，和默认界面。
如上的设置当用localhost域名访问的时候会跳转到www/localhost目录下的index.html。
而用127.0.0.1访问的时候由于没有配置映射关系会访问www目录下的index.html。
- gzip_off：设置为true则全局强制关闭gzip压缩。
- index字段可以配置为具体页面也可以配置为控制器，如 'index' => ['TestController', 'test']将会指向TestController/test，这里的test不携带前缀标示

mysql.php

mysql.php

mysql服务器配置，active是默认激活的连接

```
$config['mysql']['enable'] = true;
$config['mysql']['active'] = 'test';
$config['mysql']['test']['host'] = '127.0.0.1';
$config['mysql']['test']['port'] = '3306';
$config['mysql']['test']['user'] = 'root';
$config['mysql']['test']['password'] = '123456';
$config['mysql']['test']['database'] = 'one';
$config['mysql']['test']['charset'] = 'utf8';
$config['mysql']['asyn_max_count'] = 10;
```

asyn_max_count

连接池内最大的连接数量。

需要注意的是SD框架每一个worker进程都包含一个连接池，比如开启了4个worker那么最大消耗mysql的连接数为40。

每一个worker最大启用10个mysql连接，如果并发要求高那么worker会尝试开启更多的mysql连接，最大为10。

redis.php

redis.php

redis服务器配置，active是默认激活的连接，可以通过代码在服务器启动时添加更多的连接池。

如果没有设置redis密码可以注释掉这一行，或者设置为空。

```
$config['redis']['local']['password'] = '';
```

```
/**
 * 选择数据库环境
 */
$config['redis']['enable'] = true;
$config['redis']['active'] = 'local';

/**
 * 本地环境
 */
$config['redis']['local']['ip'] = 'localhost';
$config['redis']['local']['port'] = 6379;
$config['redis']['local']['select'] = 1;
$config['redis']['local']['password'] = '123456';

/**
 * 本地环境2
 */
$config['redis']['local2']['ip'] = 'localhost';
$config['redis']['local2']['port'] = 6379;
$config['redis']['local2']['select'] = 2;
$config['redis']['local2']['password'] = '123456';

/**
 * 这个不要删除，dispatch使用的redis环境
 * dispatch使用的环境
 */
$config['redis']['dispatch']['ip'] = 'unix:/var/run/redis/redis.sock';
$config['redis']['dispatch']['port'] = 0;
$config['redis']['dispatch']['select'] = 1;
$config['redis']['dispatch']['password'] = '123456';

$config['redis']['asyn_max_count'] = 10;

/**
 * 最终的返回，固定写这里
 */
return $config;
```

asyn_max_count

同mysql配置中意思一样

请注意select是选择redis的库，默认有16个库，从0开始。

有开发者说通过redis-cli可以访问到，为什么框架访问不到，因为选择的库不同。redis-cli如果不使用select命令默认选择的是0库。

timerTask.php

timerTask.php

定时Task/定时Model配置文件

使用task_name将会调用Task，使用model_name将会调用Model，Task是同步阻塞，Model是异步非阻塞，如果是长时间耗时任务建议用Task。

```
/**
 * timerTask定时任务
 * （选填）task名称 task_name
 * （选填）model名称 model_name task或者model必须有一个优先匹配task
 * （必填）执行task的方法 method_name
 * （选填）执行开始时间 start_time,end_time) 格式： Y-m-d H:i:s 没有代表一直执行,一旦end_time设置后会进入1天一轮回的模式
 * （必填）执行间隔 interval_time 单位： 秒
 * （选填）最大执行次数 max_exec, 默认不限次数
 * （选填）是否立即执行 delay, 默认为false立即执行
 */
$config['timerTask'] = [];
//下面例子表示在每天的14点到20点间每隔1秒执行一次
/*$config['timerTask'][] = [
    'start_time' => 'Y-m-d 19:00:00',
    'end_time' => 'Y-m-d 20:00:00',
    'task_name' => 'TestTask',
    'method_name' => 'test',
    'interval_time' => '1',
];*/
//下面例子表示在每天的14点到15点间每隔1秒执行一次，一共执行5次
/*$config['timerTask'][] = [
    'start_time' => 'Y-m-d 14:00:00',
    'end_time' => 'Y-m-d 15:00:00',
    'task_name' => 'TestTask',
    'method_name' => 'test',
    'interval_time' => '1',
    'max_exec' => 5,
];*/
//下面例子表示在每天的0点执行1次(间隔86400秒为1天)
/*$config['timerTask'][] = [
    'start_time' => 'Y-m-d 23:59:59',
    'task_name' => 'TestTask',
    'method_name' => 'test',
    'interval_time' => '86400',
];*/
//下面例子表示在每天的0点执行1次
/*$config['timerTask'][] = [
    'start_time' => 'Y-m-d 14:53:10',
    'end_time' => 'Y-m-d 14:54:11',
```

```
'task_name' => 'TestTask',  
'method_name' => 'test',  
'interval_time' => '1',  
'max_exec' => 1,  
];*/  
return $config;
```

如果开启了Consul可以通过Consul的KV来动态修改和删除添加定时任务。

使用Task有很多限制，如果是长耗时任务可以使用自定义进程，通过TimerTask.php配置调用Model，在Model里面通过进程间通讯API调用自定义进程

log.php

log.php

日志的配置文件

现提供2种类型的日志收集策略。

- file

传统的系统文件日志

- graylog

通过graylog进行日志收集

efficiency_monitor_enable字段启动后会进行效率监控便于graylog收集分析找到效率低下的方法进行优化，file类型请不要开启，会产生大量的日志。

```
//node的名字，每一个都必须不一样
$config['log']['active'] = 'file';
$config['log']['log_level'] = \Monolog\Logger::DEBUG;
$config['log']['log_name'] = 'SD';

$config['log']['graylog']['udp_send_port'] = 12500;
$config['log']['graylog']['ip'] = '127.0.0.1';
$config['log']['graylog']['port'] = '12201';
$config['log']['graylog']['api_port'] = '9000';
$config['log']['graylog']['efficiency_monitor_enable'] = true;

$config['log']['file']['log_path'] = '/../../';
$config['log']['file']['log_max_files'] = 15;
$config['log']['file']['efficiency_monitor_enable'] = false;
return $config;
```

consul.php

consul.php

微服务的配置文件

- leader_service_name

用于同种类型的服务进行leader选举，系统启动后会告诉你是leader还是不是leader。

- watches

监听服务

- services

发布服务

这里监听和发布的都是Controller名。

```
//是否启用consul
$config['consul_enable'] = false;
//数据中心配置
$config['consul']['datacenter'] = 'dc1';
//服务器名称，同种服务应该设置同样的名称，用于leader选举
$config['consul']['leader_service_name'] = 'Test';
//node的名字，每一个都必须不一样
$config['consul']['node_name'] = 'SD-1';
//consul的data_dir默认放在临时文件下
$config['consul']['data_dir'] = "/tmp/consul";
//consul join地址，可以是集群的任何一个，或者多个
$config['consul']['start_join'] = ["192.168.8.85"];
//本地网卡地址
$config['consul']['bind_addr'] = "192.168.8.57";
//监控服务
$config['consul']['watches'] = ['MathService'];
//发布服务
// $config['consul']['services']=['MathService:8081'];
//是否开启TCP集群，启动consul才有用
$config['cluster']['enable'] = true;
//TCP集群端口
$config['cluster']['port'] = 9999;

//***断路器设置***
//阈值
$config['fuse']['threshold'] = 0.01;
//检查时间
$config['fuse']['checktime'] = 2000;
//尝试打开的间隔
$config['fuse']['trytime'] = 1000;
//尝试多少个
$config['fuse']['trymax'] = 3;
```

```
return $config;
```

集群配置

集群基于Consul，所以必须启动Consul服务，然后打开['cluster']['enable']配置好集群端口即可。

catCache.php

catCache.php

配置catCache高速缓存

```
/**
 * 是否启动CatCache
 */
$config['catCache']['enable'] = true;
//自动存盘时间
$config['catCache']['auto_save_time'] = 1000;
//落地文件夹
$config['catCache']['save_dir'] = BIN_DIR . '/cache/';
//分割符
$config['catCache']['delimiter'] = ".";
/**
 * 最终的返回, 固定写这里
 */
return $config;
```

enable

设置为true将会启动高速缓存，Actor需要高速缓存的支持。

auto_save_time

自动全盘落地的时间间隔

save_dir

存盘的路径

delimiter

访问子元素的连接符

client.php

client.php

配置client相关的参数

```
$config['httpClient']['asyn_max_count'] = 10;
$config['tcpClient']['asyn_max_count'] = 10;

$config['tcpClient']['test']['pack_tool'] = 'LenJsonPack';

//默认用于consul微服务的rpc配置
$config['tcpClient']['consul']['pack_tool'] = 'LenJsonPack';
//不同服务指定不同解析策略
$config['tcpClient']['consul_MathService']['pack_tool'] = 'LenJsonPack';
return $config;
```

httpClient

- asyn_max_count配置HttpClientPool的最大连接数

tcpClient

- test 对应test名称的配置
- consul 框架consulRPC服务的默认配置
- consul_MathService 框架consulRPC服务的MathService的配置

配置需要提供pack_tool，每个配置应用一个封装器。

自定义配置

自定义配置

在配置文件夹中可以创建一个新的php文件作为新配置。

规范

```
<?php
$config['myConfig']['config1'] = "aaaa";
return $config;
```

- 必须提供一个数组，数组中存放相应的配置。
- 这个数组是和别的配置通用的，所以建议使用多维数组。
- 键名不允许出现 “.” 。
- 文件名没有实际意义
- 文件末尾必须返回这个数组，否则会报错。

获取配置

1. 通过数组的方式获取

```
$data = get_instance()->config['myConfig']['config1'];
```

2. 通过get获取

```
$data = get_instance()->config->get('myConfig.config1','bbb');
```

- 通过get方式获取可以提供默认值
- 多级间用 “.” 分割

框架入口

框架入口

- [框架入口](#)
 - [__construct](#)
 - [getEventControllerName](#)
 - [getConnectMethodName](#)
 - [getCloseMethodName](#)
 - [onOpenServiceInitialization](#)
 - [clearState](#)
 - [initAsynPools](#)
 - [beforeSwooleStart](#)
 - [startProcess](#)
 - [onWebSocketHandCheck](#)
 - [get_instance\(\)](#)
 - [其他方法](#)

AppServer是SD框架的入口类，它管理着框架启动的所有准备工作，这个类不会被Reload重载，也就是说如果修改了此类想让其生效，那么必须重新启动服务器通过Reload的方法是无法实现代码热更新的。

AppServer继承SwooleDistributedServer类，SwooleDistributedServer绝大部分的函数作为使用者来说不是很常用，接下来我们介绍些常用的方法。

__construct

可以在这定义自定义Loader

```
/**
 * 可以在这里自定义Loader，但必须是ILoader接口
 * AppServer constructor.
 */
public function __construct()
{
    $this->setLoader(new Loader());
    parent::__construct();
}
```

getEventControllerName

设置默认的事件控制器名称，长连接（TCP，WS，WSS）需要配置
通过此函数可以将客户端的connect和close的回调路由到控制器中。

```
/**
 * @return string
 */
public function getEventControllerName()
{
    return 'AppController';
}
```

上面代码将会将客户端的连接和断开事件路由到AppController控制器中。

可通过Ports配置覆盖此设置

getConnectMethodName

设置默认的事件Connect方法名称，长连接（TCP，WS，WSS）需要配置

```
/**
 * @return string
 */
public function getConnectMethodName()
{
    return 'onConnect';
}
```

客户端连接的信息会路由到AppController中onConnect方法。
如果在business.php配置中设置了前缀，控制器方法也需要包含前缀。

可通过Ports配置覆盖此设置

getCloseMethodName

设置默认的事件Close方法名称，长连接（TCP，WS，WSS）需要配置

```
/**
 * @return string
 */
public function getCloseMethodName()
{
    return 'onClose';
}
```


客户端关闭连接的信息会路由到AppController中onClose方法。
如果在business.php配置中设置了前缀，控制器方法也需要包含前缀。

可通过Ports配置覆盖此设置

onOpenServiceInitialization

这个方法是服务器启动时第一时间调用的方法，并且内部实现了锁的操作，无论开多少个进程只会有一个进程会执行该方法，并且无论怎么reload也只会启动时执行仅仅一次。

我们可以在这个方法内实现开服初始化的工作。

clearState

这个方法和onOpenServiceInitialization类似也是开服的初始化工作，但区别在于这是在服务器启动前做的操作，所以并不支持任何异步属性，如果你操作mysql只能使用pdo的方式，redis也只能使用同步扩展的方式。

initAsynPools

这里主要是进行异步客户端的初始化工作，onSwooleWorkerStart后会调用initAsynPools进行客户端的初始化。开发者可以在此函数中添加自定义的客户端，比如额外的redis，额外的mysql，mqtt，httpclient等。

```
$this->addAsynPool('GetIPAddress', new HttpClientPool($this->config, 'http://int.dpool.sina.com.cn'));
```

用法很简单通过addAsynPool添加一个署名的客户端实例。

在Controller和Model中可以通过get_instance()->getAsynPool('GetIPAddress')获取名为GetIPAddress的实例。

beforeSwooleStart

这里是进阶用法，熟悉swoole的开发者如果想创建自己的进程或者共享Table或者开放更多的端口，那么可以在这个函数中处理，名字表示的含义很清晰这是在swoole服务start方法之前调用的。
查看SwooleDistributedServer的beforeSwooleStart你会发现更多用法。

```
$this->uid_fd_table = new \swoole_table(65536);  
$this->uid_fd_table->column('fd', \swoole_table::TYPE_INT, 8);  
$this->uid_fd_table->create();
```

startProcess

启动用户进程，在这里可以自定义用户进程

```
/**
 * 用户进程
 */
public function startProcess()
{
    parent::startProcess();
    //ProcessManager::getInstance()->addProcess(MyProcess::class);
}
```

onWebSocketHandCheck

```
/**
 * 可以在这验证WebSocket连接,return true代表可以握手, false代表拒绝
 * @param HttpInput $httpInput
 * @return bool
 */
public function onWebSocketHandCheck(HttpInput $httpInput)
{
    return true;
}
```

get_instance()

get_instance()是个帮助函数，他始终返回的是SwooleDistributedServer实例，可以在代码任何位置访问到。

其他方法

- getMysql 获取同步mysql
- getRedis 获取同步redis
- sendToAllWorks 发送给所有的进程，\$callStaticFuc为静态方法,会在每个进程都执行
- sendToAllAsynWorks 发送给所有的异步进程，\$callStaticFuc为静态方法,会在每个进程都执行
- sendToRandomWorker 发送给随机进程
- sendToOneWorker 发送给指定进程
- isReload 是否是重载
- sendToAll 广播
- sendToUid 向uid发送消息

- pubToUid 向uid发布消息
- sendToUids 批量发送消息
- getSubMembersCountCoroutine 获取Topic的数量
- getSubMembersCoroutine 获取Topic的Member
- getUidTopicsCoroutine 获取uid的所有订阅
- addSub 添加订阅
- removeSub 移除订阅
- pub 发布订阅
- addAsynPool 添加异步连接池
- getAsynPool 获取连接池
- isCluster 是否开启集群
- isConsul 是否开启Consul
- kickUid 踢用户下线
- bindUid 将fd绑定到uid,uid不能为0
- unBindUid 解绑uid, 链接断开自动解绑
- coroutineUidIsOnline uid是否在线
- coroutineCountOnline 获取在线人数
- coroutineGetAllUids 获取所有在线uid
- stopTask 向task发送强制停止命令
- getServerAllTaskMessage 获取服务器上正在运行的Task
- getBindIp 获取本机ip
- getUidInfo 获取uid信息

MVC架构

MVC架构

MVC全名是Model View Controller，是模型(model) - 视图(view) - 控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

MVC 是一种使用 MVC (Model View Controller 模型-视图-控制器) 设计创建 Web 应用程序的模式：

- Model (模型) 表示应用程序核心 (比如数据库记录列表)。
- View (视图) 显示数据 (数据库记录)。
- Controller (控制器) 处理输入 (写入数据库记录)。
-

Model (模型) 是应用程序中用于处理应用程序数据逻辑的部分。

通常模型对象负责在数据库中存取数据。

View (视图) 是应用程序中处理数据显示的部分。

通常视图是依据模型数据创建的。

Controller (控制器) 是应用程序中处理用户交互的部分。

通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC 分层有助于管理复杂的应用程序，因为您可以在一个时间内专门关注一个方面。例如，您可以在不依赖业务逻辑的情况下专注于视图设计。同时也让应用程序的测试更加容易。

MVC 分层同时也简化了分组开发。不同的开发人员可同时开发视图、控制器逻辑和业务逻辑。

SD框架中的MVC

SD根目录中有个app文件夹，该文件夹作为业务逻辑存放的主文件夹。

- app/Controllers 存放控制器代码
- app/Models 存放模型代码
- app/Views 存放视图代码
- app/Tasks 存放同步任务代码

Controllers和Models下开发者可以继续细分文件夹，但要满足psr-4的规范，即代码的命名空间要与代码的存放目录一一对应。

在设计上Models目录下可以继续划分为业务层代码和数据层代码。

调用关系如下：

Controller可以调用Model和Task，Model可以调用Model和Task，Task可以有选择的调用部分Model。

Task为同步代码，如果Model中使用了异步客户端那么Task则不能调用该Model，框架对Redis和Mysql客户端进行了处理，可以自动切换在worker中使用异步在task中使用同步，所以Model中仅仅使用了redis或者mysql是可以在task中调用的。

View视图，仅仅在Http中需要被使用，view本质上是模板文件。

加载器-Loader

加载器-Loader

- [加载器-Loader](#)
 - [自定义加载器](#)
 - [model](#)
 - [task](#)
 - [view](#)

在Controller , Model , Task中经常用到。

自定义加载器

可以自定义加载器，需要实现ILoader接口，然后在AppServer的__construct方法中注入。

```
/**
 * 可以在这里自定义Loader，但必须是ILoader接口
 * AppServer constructor.
 */
public function __construct()
{
    $this->setLoader(new Loader());
    parent::__construct();
}
```

model

通过加载器加载并返回一个model的实例。

函数原型

```
/**
 * 获取一个model
 * @param $model string
 * @param $parent CoreBase
 */
function model($model, $parent)
```

其中\$model是Model的类名，根据SD的传统该类优先在app/Models中寻找，如果不存在则在Server/Models中寻找。

\$parent是调用的容器，一般都是传入\$this。

例子：

```
public function test_model()
{
    $testModel = $this->loader->model('TestModel',$this);
    $testModel->timerTest();
}
```

Model是专门和数据打交道的模块。

task

通过加载器加载并返回一个task的代理。

函数原型

```
/**
 * 获取一个task
 * @param $task
 * @return mixed|null|TaskProxy
 * @throws SwooleException
 */
public function task($task)
```

其中\$task是Task的类名，根据SD的传统该类优先在app/Tasks中寻找，如果不存在则在Server/Tasks中寻找。

例子：

```
public function test_task()
{
    $testTask = $this->loader->task('TestTask');
    $result = $testTask->test();
}
```

TestTask有个test方法，虽然说\$testTask是个TaskProxy，但你可以把他当做是TestTask调用方法。

view

通过加载器加载并返回一个模板

函数原型

```
/**
 * view 返回一个模板
```

```
* @param $template
* @param array $data
* @param array $mergeData
* @return string
*/
public function view($template, $data = [], $mergeData = [])
```

例子：

```
/**
 * html测试
 */
public function http_html_test()
{
    $template = $this->loader->view('server::404');
    $this->http_output->end(template);
}
```


控制器-Controller

控制器-Controller

Controller需要遵循的规则：

- 1.控制器需要继承Controller类。
- 2.控制器里所有public方法都是公开给外界访问的，private和protected定义的方法无法被外界访问。
- 3.控制器里的方法需要搭配前缀名，具体的前缀名的定义在ports.php配置中。
- 4.控制器采用了对象池复用技术，需要注意调用destroy()方法，否则会产生内存泄露。

特别声明：从3.1.1版本起destroy方法会自动调用，无需手动书写。

initialization

初始化方法，Controller是对象池复用模式，所以不要在下划线construct中进行初始化，通过重写initialization方法实现Controller的初始化，每次路由到Controller中时都会执行initialization方法。

```
class AppController extends Controller
{
    /**
     * @var AppModel
     */
    public $AppModel;

    protected function initialization($controller_name, $method_name)
    {
        parent::initialization($controller_name, $method_name);
        $this->AppModel = $this->loader->model('AppModel', $this);
        throw new \Exception("错误");
    }
}
```

可以在initialization中进行loader操作，也可以进行细节的检查，如果想中断操作直接抛出异常即可，会直接进入异常处理函数，不会执行后面的方法。

特别声明：从3.1.1版本起增加了interrupt方法专门用于中断

interrupt

3.1.1版本新增方法

中断，调用此方法Controller立马进入销毁，后面的代码一概不会执行。

```
protected function initialization($controller_name, $method_name)
{
    parent::initialization($controller_name, $method_name);
    $this->AppModel = $this->loader->model('AppModel', $this);
    $this->interrupt();
}
```

defaultMethod

当控制器方法不存在的时候的默认方法，开发者可以重写此方法作为默认方法

onExceptionHandler

onExceptionHandler 异常的回调，当Controller方法中出现异常会转到这个回调方法中，开发者可以自定义异常的处理和输出。

在任何方法中包括调用的Model，Task抛出异常最终都会中断流程，并且流入到onExceptionHandler方法中

HTTP

列举下对应的配置

port.php

```
$config['ports'][] = [
    'socket_type' => PortManager::SOCK_HTTP,
    'socket_name' => '0.0.0.0',
    'socket_port' => 8081,
    'route_tool' => 'NormalRoute',
    'middlewares' => ['MonitorMiddleware', 'NormalHttpMiddleware'],
    'method_prefix' => 'http_'
];
```

ports配置中配置了8081端口为HTTP协议，路由使用了NormalRoute路由，中间件使用了MonitorMiddleware和NormalHttpMiddleware，方法前缀为'http_'。

关于路由和中间件，请看相应的文档，这里我们对应下url和目标Controller的关系。

URL：<http://localhost:8081/TestController/echo>

Controller文件：将会优先寻找app/Controllers/TestController.php，如果找不到会寻找框架里的Server/Controllers/TestController.php

在server.php配置中有个allow_ServerController参数，如果为false将不能访问到框架自带的控制器，Controller，Model，Task，Middlewares，Route，Pack均会优先从app目录下匹配。

由于设置了method_prefix，那么我们访问的方法就是TestController类中的http_echo。

我们接下来看看http_echo方法，代码如下：

```
public function http_echo()
{
    $this->http_output->end(123);
}
```

通过\$this->http_output->end方法进行输出，通过浏览器访问可以看到123的字样。

end方法只能调用一次，多次调用无效。

通过\$this->http_output可以获取到HttpOutputPut类，里面有相应的API，这里不多做介绍。
相应的通过\$this->http_input可以获取到HttpInputPut类。

```
public function http_test()
{
    $max = $this->http_input->get('max');
    if (empty($max)) {
        $max = 100;
    }
    $sum = 0;
    for ($i = 0; $i < $max; $i++) {
        $sum += $i;
    }
    $this->http_output->end($sum);
}
```

这里通过\$this->http_input->get方法获取到了max的值。

访问的URL为http://localhost:8081/TestController/test?max=10

将输出45.

- redirect Http重定向
- redirect404 重定向到404
- redirectController 重定向到控制器，这里的方法名不填前缀

TCP

列举下对应的配置

port.php

```
$config['ports'][] = [
    'socket_type' => PortManager::SOCK_TCP,
    'socket_name' => '0.0.0.0',
```

```
'socket_port' => 9091,
'pack_tool' => 'LenJsonPack',
'route_tool' => 'NormalRoute',
'middlewares' => ['MonitorMiddleware']
];
```

这里开启了9091端口作为TCP协议，封装器使用了LenJsonPack，路由器使用了NormalRoute，中间件为MonitorMiddleware，大家先不要对这么多名称感到恐惧，我们后面慢慢来了解。

HTTP协议是个完整的业务层协议所以他不需要封装器的支持，只需要进行路由，但TCP协议不同一般我们会在TCP协议上再构建一个私有的协议，这时我们就需要封装器了。具体的封装器我们以后详细说明，现在我们假定我们通过了路由将消息转发到了对应的控制器方法上了。

这里没有设置前缀所以我们直接路由到对应的方法上。

```
/**
 * tcp的测试
 */
public function testTcp()
{
    var_dump($this->client_data->data);
    $this->send($this->client_data->data);
}
```

通过\$this->client_data可以获取到数据流通过封装器解包后的完整数据，NormalRoute中我们定义client_data的结构如下

```
stdClass{
    controller_name:控制器名称
    method_name:方法名称
    path:路径（非必须）
    params:方法参数（非必须）
    *:等等其他自定义字段
}
```

client_data是个object不是一个array这点需要注意，数据流通过封装器解包后传递给路由器，路由器需要controller_name和method_name，如果满足条件将进行路由。

上面的代码将获取到的\$this->client_data->data返回给了客户端。

返回的数据将又一次经过端口设置的封装器，封装成协议支持的格式返回给客户端。

```
public function testTcp()
{
    $this->send($this->client_data->data)); //发送第一条
    $this->send($this->client_data->data)); //发送第二条
}
```

send系列方法被用于长连接（tcp，ws，wss）发送消息。

- send 向当前客户端发送消息
- sendToUid 向指定uid发送消息
- sendToUids 向指定uids发送消息
- sendToAll 向所有绑定uid的连接发送消息
- bindUid 为当前连接绑定uid
- kickUid 踢用户下线
- addSub 添加订阅
- removeSub 移除订阅
- sendPub 发布订阅
- getFdInfo 获取fd的信息

WebSocket

和TCP基本一致，贴一下对应的配置

```
$config['ports'][] = [
    'socket_type' => PortManager::SOCK_WS,
    'socket_name' => '0.0.0.0',
    'socket_port' => 8083,
    'route_tool' => 'NormalRoute',
    'pack_tool' => 'NonJsonPack',
    'opcode' => PortManager::WEBSOCKET_OPCODE_TEXT,
    'middlewares' => ['MonitorMiddleware', 'NormalHttpMiddleware']
];
```

opcode是WebSocket独有的，有2个选项

PortManager::WEBSOCKET_OPCODE_TEXT 文本流

PortManager::WEBSOCKET_OPCODE_BINARY 二进制流

其中还需要注意下封装器使用的是NonJsonPack，NonJsonPack和LenJsonPack区别在于NonJsonPack没有长度信息，因为ws协议也是一个相对上层的协议不要手动分割数据流。

模型-Model

Model

可以在此层级上划分出业务层和数据层，开发者可以根据自己的喜好来创建相应的文件夹。

比如app/Models/Data/这个目录下存放数据层，app/Models/Business/这个目录存放业务层。

Loader

可以通过loader加载其他的model，如果是多层model可以直接包含文件夹名称进行loader，也可以是用class进行loader。

```
$this->loader->model("Data/TestModel",$this);  
$this->loader->model(TestModel::class,$this);
```

使用数据库

```
$result = $this->redis_pool->getCoroutine()->get('testroute');  
$result = $this->db->select('*')  
    ->from('account')->limit(1)  
    ->coroutineSend();
```

__construct

对象池模式Model会被复用，整个生命周期中__construct只会执行一次。

```
class TestModel extends Model  
{  
  
    public function __construct()  
    {  
        parent::__construct(TestModelProxy::class);  
    }  
}  
class TestModelProxy extends ChildProxy  
{  
    public function test_exception()  
    {  
        $this->beforeCall("test_exception");  
        return $this->own->test_exception();  
    }  
}
```

上面是高级用法，设置AOP代理，默认每个Model都有层AOP代理，这里可以自定义AOP代理。

initialization

通过重写initialization进行Model的初始化，每次Loader Model的时候都会执行initialization方法，不建议在__construct进行初始化因为是对象池模式Model会被复用，而在整个生命周期中__construct只会执行一次。

获取异步连接池

首先需要在AppServer中创建连接池

```
public function initAsynPools()
{
    parent::initAsynPools();
    $this->addAsynPool('GetIPAddress', new HttpClientPool($this->config, 'http://int.d
pool.sina.com.cn'));
    $this->addAsynPool('WeiXinAPI', new HttpClientPool($this->config, 'https://api.wei
xin.qq.com'));
}
```

然后在Model的initialization方法中获取

```
protected $GetIPAddressHttpClient;
public function initialization()
{
    parent::initialization();
    $this->GetIPAddressHttpClient = get_instance()->getAsynPool('GetIPAddress');
}
```

Mysql多库就是创建多个Mysql连接池实现的，但是Mysql连接池需要多一个步骤

```
public function initialization()
{
    parent::initialization();
    $this->mysql2 = get_instance()->getAsynPool('mysql2');
    $this->installMysqlPool($this->mysql2);
}
```

视图-View

View

通过加载器加载并返回一个模板

loader

```
/**
 * html测试
 */
public function http_html_test()
{
    $template = $this->loader->view('server::404');
    $this->http_output->end($template);
}
```

server::404 是指Server/Views下的404.blade.php文件

app::404 是指app/Views下的404.blade.php文件

具体的请参考blade模板引擎。

修改模板引擎

SD支持自定义模板引擎。

可以通过重写AppServer中的setTemplateEngine方法设置自定义的模板引擎，以下是设置Plates模板引擎的代码。

```
/**
 * 设置模板引擎
 */
public function setTemplateEngine()
{
    $this->templateEngine = new Engine();
    $this->templateEngine->addFolder('server', __DIR__ . '/Views');
    $this->templateEngine->addFolder('app', __DIR__ . '/../app/Views');
    $this->templateEngine->registerFunction('get_www', 'get_www');
}
```

再通过自定义Loader实现自定义的view加载即可。

Blade教程

同步任务-Task

同步任务-Task

task用作执行一些耗时操作，或者同步客户端，task本身是同步的，不能使用异步方法，task中也不能调用task。

创建一个Task

继承Task类。

```
class TestTask extends Task
{
    public function test_task()
    {
        $testModel = $this->loader->model('TestModel', $this);
        $result = $testModel->test();
        print_r($result);
    }
}
```

Task中可以调用Model，但是该Model有限制，不能有异步的API命令（redis，mysql框架会自动转换为同步模式，所以可以使用）

调用一个Task

```
$testTask = $this->loader->task(TestTask::class, $this);
$result = $testTask->test();
```

可以直接访问Task，task的执行是异步的，不会堵塞worker进程。

也可以设置更丰富的协程属性

```
$testTask->call("test", [], -1, function (TaskCoroutine $taskCoroutine){
    $taskCoroutine->setTimeout(1000);
});
```

也可以使用回调

```
$testTask->startTask("test", [], -1, function ($result){
    //result是结果
});
```

上面的方法的回调函数是异步执行的，那时可能调用这个task的控制器和模型已经被销毁了。

如果仅仅是通知Task执行某个任务不想获得结果可以使用startTask方法，并将回调函数设置为null。

限制

Task是同步进程，限制很多：

- 1.不能使用异步客户端
- 2.不能使用进程间通信

Sub/Pub基于进程间通信实现的，所以不能在Task中使用

封装器

封装器

了解封装器前，请先了解Swoole支持的协议类型

Swoole编程指南-EOF协议

Swoole编程指南-EOF协议

[原文链接](#)

- [Swoole编程指南-EOF协议](#)
 - [什么是EOF协议](#)
 - [开启EOF支持](#)
 - [实战](#)

什么是EOF协议

EOF (End of File) 是一个结束标记，意思为在逐个读取数据流中的数据时，如果发现读到EOF标记，就代表已经读到了数据末尾。在TCP的数据流中，使用EOF协议的数据流特征如下：

| 数据 | EOF | 数据 | EOF |

在每一串正常数据的末尾，添加一个预先规定的、绝对不会出现在数据中的字符串作为结束标记。这样接收到的数据就可以根据这个EOF标记来切分数据。

开启EOF支持

在Swoole中，可以使用如下配置选项来开启EOF功能：

```
$server->set([
    'open_eof_split' => true,    // 开启EOF检测
    'package_eof' => '/r/n' ,    // 设置EOF标记
]);
```

其中，open_eof_split选项会开启Swoole底层对接收到的数据从头开始依次扫描检查，当找到第一个EOF标记时，将已经扫描过的数据作为一个完整的数据包通过onReceive回调发送给PHP层处理。这里需要注意的是，package_eof只允许设置长度不超过8的字符串。

这里要注意到，open_eof_split选项是依次扫描数据中的EOF标记的，这样虽然保证每次回调都只会收到一个完整的数据包，但是性能较差。因此Swoole还提供了另外一个不同的选项：

```
$server->set([
    'open_eof_check' => true,    // 开启EOF检测
]);
```

open_eof_check同样会开启EOF检测。不同的是，open_eof_check只检查接收数据的末尾是否为EOF标记
本文档使用 [看云](#) 构建

记。相比于open_eof_split，这种方式性能最好，几乎没有损耗。但是如果同时收到了N条带有EOF标记的数据，这种方式会同时将N个数据包合并为一个回调给PHP层处理，因此需要在PHP层通过EOF标记对数据做二次拆分。

实战

为了演示效果，我将直接使用open_eof_check选项，并演示如何拆分数据包。首先是服务器端代码，如下：

```
class Server
{
    private $serv;

    public function __construct()
    {
        $this->serv = new swoole_server("0.0.0.0", 9501);
        $this->serv->set(array(
            'worker_num' => 1,
            'open_eof_check' => true,
            'package_eof' => "\r\n",
        ));

        $this->serv->on('Connect', array($this, 'onConnect'));
        $this->serv->on('Receive', array($this, 'onReceive'));
        $this->serv->on('Close', array($this, 'onClose'));

        $this->serv->start();
    }

    public function onConnect( $serv, $fd, $from_id )
    {
        echo "Client {$fd} connect\n";
    }

    public function onReceive( swoole_server $serv, $fd, $from_id, $data )
    {
        $data_list = explode("\r\n", $data);
        foreach ($data_list as $item)
        {
            if(empty($item))
                continue;
            var_dump($item);
            $serv->send($fd, $item . "\r\n");
        }
    }

    public function onClose( $serv, $fd, $from_id )
    {
        echo "Client {$fd} close connection\n";
    }
}
```

```
    }  
}  
new Server();
```

可以看到，在接收到数据后，我先将接收到的数据用EOF标记作为分隔符做了一次拆分，这样就能得到一个个独立的完整数据包，然后依次处理即可。

Swoole编程指南-固定包头协议

Swoole编程指南-固定包头协议

[原文链接](#)

- [Swoole编程指南-固定包头协议](#)
 - [什么是固定包头协议](#)
 - [知识科普：字节序](#)
 - [开启协议检测](#)
 - [实战](#)

什么是固定包头协议

固定包头协议是在需要发送的数据前，加上一段预先规约好长度和格式的数据体，在该数据体中存放有后续数据的相应信息（一般情况下，后续数据的长度字段是必填项）的协议，这样一段数据体则称之为协议包头。在TCP的数据流中，使用固定包头协议的数据流特征如下：

| len长度 | 数据 | len长度 | 数据 |

知识科普：字节序

这里需要说明一下字节序的问题。在内存中，一个整数由四个连续的字节序列表示。例如一个int类型的变量x，其内存地址为0x100，那么这个变量的四个字节会被存储在0x100，0x101，0x102，0x103。字节序则决定了这个变量的内容在这四个地址中的存放顺序。通常根据字节排列顺序的不同，将字节序区分为大端字节序和小端字节序。这里假设该变量x的16进制表示为0x12345678，在大端字节序中，该变量将以如下形式存放在内存中：

... 0x100 0x101 0x102 0x103 ...

... 12 34 56 78 ...

而在小端字节序中，则以如下形式存放：

... 0x100 0x101 0x102 0x103 ...

... 78 56 34 12 ...

可以看到，以内存地址从小到大代表从低到高，并且将变量从左到右设定为从低到高，大端序列就是高位字节存放在高位，低位字节存放在低位，而小端序反之。

在机器中，使用大端序和小端序完全取决于硬件本身。因此，在不同的机器之间通信时，需要有一个统一的字节序来进行传输，由此诞生了网络字节序的概念。网络字节序一般采用大端序（IP协议指定），而机器字节序则由机器本身决定。两者之间需要借助系统函数进行转换。

开启协议检测

在Swoole中，可以使用如下配置选项来开启固定包头协议功能：

```
$server->set([
    'open_length_check'      => 1,          // 开启协议解析
    'package_length_type'    => 'N',        // 长度字段的类型
    'package_length_offset'  => 0,          //第N个字节是包长度的值
    'package_body_offset'    => 4,          //第N个字节开始计算长度
    'package_max_length'     => 20000000,   //协议最大长度
]);
```

在这里，package_length_type规定了length长度字段的类型，这个类型等价于使用PHP的pack函数打包数据时使用的类型，具体类型如下表所示:

类型	长度	说明	范围
c	1字节	有符号Char	-128 ~ 127
C	1字节	无符号Char	0 ~ 256
s	2字节	有符号，机器字节序	-32768 ~ 32767
S	2字节	无符号，机器字节序	0 ~ 65535
n	2字节	无符号，网络字节序	0 ~ 65535
N	4字节	无符号，网络字节序	0 ~ 4294967295
l	4字节	有符号，机器字节序	-2147483648 ~ 2147483648
L	4字节	无符号，机器字节序	0 ~ 4294967295
v	2字节	无符号，小端字节序	0 ~ 65535
V	4字节	无符号，小端字节序	0 ~ 4294967295

实战

首先，我们规定如下的协议格式：

序号	len长度	数据
4字节	4字节	不定长

其中，序号为从0开始的自增序号，返回的数据也带有同样的序号用于标记同一次请求。长度字段为后续数据的实际长度，因此，一个完整请求的长度为length + 8。服务器每次接收到的数据中，从第四个字节开始是长度字段，类型为4字节无符号整型。从第八个字节开始是数据体，接收length个字节后结束，记

为一个完整数据包。

服务器设置代码如下所示：

```
$serv->set(array(
    'worker_num' => 1,
    'open_length_check' => true,          // 开启协议解析
    'package_length_type' => 'N',        // 长度字段的类型
    'package_length_offset' => 4,         // 第4个字节开始是包长度
    'package_body_offset' => 8,          // 第8个字节开始计算长度
    'package_max_length' => 20000000,    // 协议最大长度
));
```

接收数据部分代码如下：

```
public function onReceive( swoole_server $serv, $fd, $from_id, $data )
{
    $head = unpack('NN', substr($data, 0 , 8)); // 获取包头
    $body = substr($data, 8);                  // 获取数据
    var_dump($body);
    $response = "Hello";
    $num = $head[0];
    $len = strlen($response);

    $content = pack('NN', $num, $len) . $response; // 拼接响应内容
    $serv->send($fd, $content);                    // 发送数据
}
```

封装器-Pack

封装器-Pack

```
interface IPack
{
    function encode($buffer);

    function decode($buffer);

    function pack($data, $topic = null);

    function unPack($data);

    function getProbufSet();

    function errorHandle($e, $fd);
}
```

需要实现6个方法

encode和decode分别是协议头的封装和解析工作，pack和unPack是协议体的序列化和反序列化工作，getProbufSet将返回swoole的set中关于协议头解析相关的配置，errorHandle是协议解析出错后的处理函数。

默认提供了几组简易配置

```
class LenJsonPack implements IPack
{
    protected $package_length_type = 'N';
    protected $package_length_type_length = 4;
    protected $package_length_offset = 0;
    protected $package_body_offset = 0;

    protected $last_data = null;
    protected $last_data_result = null;

    /**
     * 数据包编码
     * @param $buffer
     * @return string
     * @throws SwooleException
     */
    public function encode($buffer)
    {
        $total_length = $this->package_length_type_length + strlen($buffer) - $this->package_body_offset;
```

```

        return pack($this->package_length_type, $total_length) . $buffer;
    }

    /**
     * @param $buffer
     * @return string
     */
    public function decode($buffer)
    {
        return substr($buffer, $this->package_length_type_length);
    }

    public function pack($data, $topic = null)
    {
        if ($this->last_data != null && $this->last_data == $data) {
            return $this->last_data_result;
        }
        $this->last_data = $data;
        $this->last_data_result = $this->encode(json_encode($data, JSON_UNESCAPED_UNICODE));
        return $this->last_data_result;
    }

    public function unPack($data)
    {
        $value = json_decode($this->decode($data));
        if (empty($value)) {
            throw new SwooleException('json unPack 失败');
        }
        return $value;
    }

    public function getProbufSet()
    {
        return [
            'open_length_check' => true,
            'package_length_type' => $this->package_length_type,
            'package_length_offset' => $this->package_length_offset,          //第N个字节
            'package_body_offset' => $this->package_body_offset,             //第几个字节开始
            'package_max_length' => 20000000,    //协议最大长度)
        ];
    }

    public function errorHandle(\Throwable $e, $fd)
    {
        get_instance()->close($fd);
    }
}

```

是包长度的值
计算长度

自定义协议举例

协议 = 包头(4字节 msg_type + 4字节包体长) + 包体(protobuf)

```
<?php

namespace app\Pack;

use Server\Pack\IPack;
use Server\CoreBase\SwooleException;
use Game\Protobuf\Ping;
use Game\Protobuf\Pong;
use stdClass;

class GamePack implements IPack
{
    /**
     * 数据包编码
     * @param $buffer
     * @return string
     * @throws SwooleException
     */
    public function encode($buffer)
    {
        return substr($buffer, 0, 4) . pack('N', strlen($buffer) - 4) . substr($buffer, 4);
    }

    /**
     * @param $buffer
     * @return string
     */
    public function decode($buffer)
    {
        return substr($buffer, 0, 4) . substr($buffer, 8);
    }

    // 包长还是会传递过来
    public function unPack($data)
    {
        $msgType = unpack('N', substr($data, 0, 4))[1];
        $body = substr($data, 8);

        $dataMsg = new stdClass();
        $dataMsg->msg_type = $msgType;
        // protobuf Exception
        // try {
        //     $to->mergeFromString($data);
        // } catch (Exception $e) {
        //     // Handle parsing error from invalid data.
        //     ...
    }
}
```

```
//    }
    if ($msgType == 1) {
        $msg = new Ping();
        $msg->mergeFromString($body);
        $dataMsg->uid = $msg->getUid();
    } else if ($msgType == 2) {
        $msg = new Pong();
        $dataMsg->uid = $msg->getUid();
    }
    return $dataMsg;
}

public function Pack($data, $topic = null)
{
    $msgType = $data->msg_type;
    if ($msgType == 1) {
        $msg = new Ping();
        $msg->setUid($data->uid);
    } else if ($msgType == 2) {
        $msg = new Pong();
        $msg->setUid($data->uid);
    }
    $data = $msg->serializeToString();
    return pack('N', $msgType) . pack('N', strlen($data)). $data;
}

public function getProbufSet()
{
    return [
        'open_length_check'      => true,
        'package_length_type'    => 'N',
        'package_length_offset' => 4,          //第N个字节是包长度的值
        'package_body_offset'    => 8,          //第几个字节开始计算长度
        'package_max_length'     => 20000000,   //协议最大长度
    ];
}

public function errorHandle(\Throwable $e, $fd)
{
    get_instance()->close($fd);
}
}
```

protobuf 定义如下:

```
syntax = "proto3";

package game.protobuf;

enum MsgType {
    PING = 1;
```

```
PONG = 2;
};

message Ping {
    uint64 uid = 1;
};

message Pong {
    uint64 uid = 1;
};
```

路由器

路由器

- 路由器
 - 多级路由
 - 自定义路由
 - 自定义协议配置 route 举例

<http://localhost:8081/TestController/test>

以上代码会先在/app/Controllers目录下寻找TestController控制器，如果没有再去/Server/Controllers目录下寻找，如果依旧没有找到将返回404界面。

test是方法名，默认前缀名为‘http_’，可以通过修改ports.php配置设置不同端口的前缀。

所以以上url会访问到/Server/Controllers下的TestController控制器的http_test方法并输出helloworld。

```
<?php
class TestController extends Controller
{
    /**
     * http测试
     */
    public function http_test()
    {
        $this->http_output->end('helloworld', false);
    }
}
```

多级路由

默认的NormalRoute支持多级路由

<http://localhost:8081/V1/TestController/test>

可以在app/Controllers目录下添加V1目录，这样上面的URL将访问V1目录下的TestController类中的test方法。

可以添加更多级的文件夹。

自定义路由

自定义路由需要实现以下几个方法


```
interface IRoute
{
    function handleClientData($data);

    function handleClientRequest($request);

    function getControllerName();

    function getMethodName();

    function getParams();

    function getPath();

    function errorHandle(\Throwable $e, $fd);

    function errorHttpHandle(\Throwable $e, $request, $response);
}
```

1. (仅仅TCP) handleClientData 设置反序列化后的数据 Object
2. (仅仅HTTP) handleClientRequest 处理http request
3. getControllerName 获取控制器名称
4. getMethodName 获取方法名称
5. (仅仅HTTP) getPath 获取url_path
6. (仅仅TCP) getParams 获取参数/扩展

解析错误的回调

```
function errorHandle(\Throwable $fd)
```

注意getParams是作为一个扩展，如果这里被返回了参数，那么这个参数会被直接当做调用Controller方法的传入参数。

```
class ProtoController extends Controller
{
    public function makeMessageData(AbstractMessage $responseMessage)
    {
        //这里的$responseMessage就是getParams()获取到的对象
    }
}
```

自定义协议配置 route 举例

和 Pack 中的例子对应, 将不同 `msg_type` 的消息分发到控制器中不同的方法下

```
namespace Server\Route;

use Server\CoreBase\SwooleException;

class GameRoute implements IRoute
{
    // 其他方法都可以保持不变

    /**
     * 获取控制器名称
     * @return string
     */
    public function getControllerName()
    {
        return 'GameController';
    }

    /**
     * 获取方法名称
     * @return string
     */
    public function getMethodName()
    {
        $methodName = 'ping';
        $msgType = $this->client_data->msg_type;
        if ($msgType == 2) {
            $methodName = 'pong';
        }
        return $methodName. 'Msg';
    }
}
```

这样, 不同类型的消息就分发到 `GameController` 下的对应 `method`, 使用 `$this->client_data` 即可获取到 `unpack` 后的数据

TCP相关

TCP相关

Controller中存在大量的tcp相关的API。

`get_instance()`将返回一个Server对象，里面也有很多tcp相关的高级API。

绑定UID

绑定UID

集群环境中fd是可以重复的，uid在集群中是唯一的。
我们通过bindUid建立fd和uid之间的一一对应关系。

bindUid

将当前连接绑定UID，会先自动执行kickUid，可以设置\$isKick参数。

kickUid

将绑定这个UID的客户端踢下线。

unBindUid

移除绑定，一般客户端断开连接后会自动调用unBindUid

close

服务器主动关闭链接

Send系列

Send系列

Send系列函数服务于TCP/WS这种长连接。

Controller中的send方法后面往往都有个\$destroy参数，默认为true，如果一旦设置为true就代表该控制器结束使命了，后面不允许再出现任何逻辑代码，如果继续send会导致异常。想发送多个信息需要先设置\$destroy为false，在最后一次调用send时设置为true。

send

向当前客户端发送消息

sendToUid

向绑定uid的客户端发送消息

sendToUids

向多个绑定uid的客户端发送消息

sendToAll

向所有绑定uid的客户端发送消息

sendToAllFd

向所有的客户端发送消息

> sendToAll是向所有绑定uid的客户端发送消息，没有绑定的不会收到消息推送。

Sub/Pub

Sub/Pub

订阅/发布，支持集群。

[订阅与发布](#)

获取服务器信息

获取服务器信息

通过get_instance()获取对应API，支持集群。

Controller里相关方法在get_instance()中也可以找到。

涉及到fd的只支持本地，因为在集群中只有uid是唯一的，fd是重复的

protect

保护一个fd不会被心跳断线。即使不满足心跳条件也不会被踢下线。

getFdFromUid（只支持本地）

通过uid查找fd

getUidFromFd（只支持本地）

通过fd查找uid

getServerPort（只支持本地）

查询fd对应的端口

getFdInfo

查询fd的相关信息

close（只支持本地）

服务器主动关闭fd链接

getSubMembersCountCoroutine

获取Topic的数量

getSubMembersCoroutine

获取Topic的Member

getUidTopicsCoroutine

获取uid的所有订阅

coroutineUidIsOnline

uid是否在线

coroutineCountOnline

获取在线人数

Http相关

Http相关

[HttpInput](#)

[HttpOutput](#)

HttpInput

HttpInput

获取Http访问的输入数据。

Controller通过\$this->http_input获取。

获取原生Swoole Request对象

也许你习惯了使用原生的swoole request对象，可以使用以下方式获取：

```
public function index(){
    $swoole_http_request = $this->http_input->request;
    //TODO 你可以自己获取并且解析原生swoole_http_request值
}
```

API

- postGet
优先先从post中取值，若没有再从get中取值；如果post与get中有相同的字段名，则以post值为准。
如果都不存在则返回空字符串
xss_clean:是否使用xss清理
- post
从post中取值，若不存在则返回空字符串
xss_clean:是否使用xss清理
- get
从get中取值，若不存在则返回空字符串
xss_clean:是否使用xss清理
- getPost
优先从get中取值，若没有再从post中取值；如果post与get中有相同的字段名，则以get值为准。如果都不存在则返回空字符串
xss_clean:是否使用xss清理
- getAllPostGet
获取所有的post和get，合并返回一个数组，若get和post中有相同字段post会覆盖get的值。
- getAllHeader
获取Http请求的头部所有信息。类型为数组，所有key均为小写。（实际上是返回原生的swoole http 请求头）
- getRawContent
获取原始的POST包体，用于非application/x-www-form-urlencoded格式的Http POST请求。（等

同于原生swoole http rawContent) , 此函数等同于PHP的fopen('php://input')。

- cookie
获取cookie数据
- getRequestHeader
获取头信息
- server
获取server信息，注意字段全是小写：

```
[request_method] => GET
[request_uri] => /
[path_info] => /
[request_time] => 1484215271
[request_time_float] => 1484215271.2045
[server_port] => 8081
[remote_port] => 56738
[remote_addr] => 127.0.0.1
[server_protocol] => HTTP/1.1
[server_software] => swoole-http-server
```

- getRequestMethod
获取请求的方式
- getRequestUri
获取请求的Uri
用此函数获取的Uri与server信息中的request_uri略有不同，getRequestUri会判断server中是否存在query_string，若存在则拼接返回一个完整的Uri
- getPathInfo
获取请求的Path getPathInfo 函数获取的信息与server中的path_info一致。

HttpOutput

HttpOutput

设置Http访问的输出数据。

Controller通过\$this->http_output获取

获取原生Swoole Response对象

前面HttpInput中用户可以方便的获取原生swoole request对象，同样你可以获取原生的response：

```
public function index(){
    $swoole_http_response = $this->http_output->response;
    //TODO 你可以自己获取并且解析原生swoole_http_response值
}
```

API

- setStatusHeader
设置返回的状态码
- setContentType
设置返回的Content-Type
- setHeader
设置返回的头信息
- end
设置http返回的body
output：发送的数据
gzip：是否启用压缩
destory：发送完是否自动销毁
- setCookie
设置HTTP响应的cookie信息。此方法参数与PHP的setcookie完全一致。
- endFile
发送文件

默认路由规则

默认路由规则

SD提供了一个默认路由NormalRoute，现在简单说下默认的URL规则。

简单访问

举例说明：

```
http://localhost:8081/TestController/test
```

第一步路由器将解析Url分离出TestController/test

第二步分离出TestController与test

第三步寻找TestController，大小写敏感，首先在app/Controllers目录下寻找TestController，找不到则Server/Controllers目录下寻找，如果还找不到则重定向到404。

大小写敏感，Server目录是框架目录

第四步查看8081端口有没有设置前缀，将前缀与test拼接作为方法名，比如http_test，在控制器中寻找这个方法名，找到则执行，没有找到会执行控制器的defaultMethod方法。

多级访问

举例说明：

```
http://localhost:8081/Action/TestController/test
```

第一步路由器将解析Url分离出Action/TestController/test

第二步分离出Action/TestController与test

第三步寻找TestController，大小写敏感，首先在app/Controllers/Action目录下寻找TestController，找不到则Server/Controllers/Action目录下寻找，如果还找不到则重定向到404。

第四步查看8081端口有没有设置前缀，将前缀与test拼接作为方法名，比如http_test，在控制器中寻找这个方法名，找到则执行，没有找到会执行控制器的defaultMethod方法。

可以更多分层url最后一个字段作为方法名

默认方法

举例说明：

```
http://localhost:8081/TestController
```

默认方法只支持单级访问，这里会寻找TestController控制器直接执行defaultMethod方法。

静态文件

举例说明：

```
http://localhost:8081/Index.html
```

寻找静态文件的逻辑是写在NormalHttpMiddleware中间件中，如果ports配置中去除这个中间件将不支持寻找静态文件。

判断逻辑如下：

- 1.看看是不是 “/ ”，如果是则按照business配置中寻找主页
- 2.寻找后缀名，如果有则认为是静态文件
- 3.根据目录寻找文件，如果有将按照fileHeader配置中定义的头输出，如果没有则重定向到404

<http://localhost:8081> 会通过business配置寻找主页

WebSocket相关

WebSocket相关

介绍WebSocket独有的方法

opcode

在ports.php配置中

```
'opcode' => PortManager::WEBSOCKET_OPCODE_TEXT,
```

- PortManager::WEBSOCKET_OPCODE_TEXT：文本模式
- PortManager::WEBSOCKET_OPCODE_BINARY：二进制模式

onWebSocketHandCheck

AppServer中onWebSocketHandCheck方法。

```
/**
 * 可以在这验证WebSocket连接,return true代表可以握手, false代表拒绝
 * @param HttpInput $httpInput
 * @return bool
 */
public function onWebSocketHandCheck(HttpInput $httpInput)
{
    return true;
}
```

可以在这验证WebSocket连接,return true代表可以握手, false代表拒绝。

通过\$httpInput可以获取到创建连接时的URL地址, 可以通过携带的token或者其他字段进行权限验证, 返回true代表验证通过可以握手。

setCustomHandshake

AppServer中setCustomHandshake方法。

是否自定义握手, 默认是false, 将采用默认的握手规则, 如果需要自定义则需要在AppServer的__construct方法中赋值为true。

onSwooleWSHandShake

AppServer中onSwooleWSHandShake方法。

自定义握手规则，可以通过重写此方法实现，setCustomHandshake需要设置为true。

发送消息

和TCP共用API

使用SSL

使用SSL

设置HTTPS和WSS时需要用到SSL。

编译Swoole

注意编译swoole时需要加上

```
--enable-openssl
```

打开ports.php配置文件

添加socket_ssl字段需要设置为true，需要设置ssl_cert_file和ssl_key_file。

```
$config['ports'][] = [  
    'socket_type' => PortManager::SOCK_HTTP  
    'socket_name' => '0.0.0.0',  
    'socket_port' => 8081,  
    'pack_tool' => 'LenJsonPack',  
    'route_tool' => 'NormalRoute',  
    'socket_ssl' => true,  
    'ssl_cert_file' => $ssl_dir . '/ssl.crt',  
    'ssl_key_file' => $ssl_dir . '/ssl.key',  
];
```

可以通过Nginx实现代理HTTPS和WSS

公共函数

公共函数

SD 定义了一些全局的函数，你可以在任何地方使用它们。

API

- `get_instance` 获取AppServer实例
- `getTickTime` 获取服务器运行到现在的毫秒数
- `getMillisecond` 获取当前的时间(毫秒)
- `getLuaSha1` 帮助Redis获取Lua
- `isDarwin` 是否是mac系统
- `sleepCoroutine` 协程的sleep
- `getServerIp` 通过设备号获取IP
- `getBindIp` 获取绑定的ip
- `getNodeName` 获取node的名称
- `getServerName` 获取Server的名称
- `getConfigDir` 获取配置目录
- `create_uuid` 创建uuid
- `secho` 打印debug信息
- `setTimezone` 设置时区为上海

自定义公共函数

通过Composer创建公共函数。

打开Composer.json文件在autoload中添加files：

```
"autoload": {
    "psr-4": {
        "Server\\": "src/Server",
        "app\\": "src/app",
        "test\\": "src/test"
    },
    "files": [
        "src/Server/helpers/Common.php"
    ]
},
```

上面是SD默认的帮助文件Server/helpers/Common.php，开发者可以在app目录下创建自己的帮助文件，最后执行Composer命令

```
composer install
```

进阶篇

内核优化

对象池

上下文-Context

中间件

进程管理

异步连接池

日志工具-GrayLog

微服务-Consul

集群-Cluster

高速缓存-CatCache

万物-Actor

消息派发-EventDispatcher

延迟队列-TimerCallBack

协程

订阅与发布

MQTT简易服务器

AMQP异步任务调度

自定义命令-Console

特别注意事项

内核优化

内核优化

- 内核优化
 - ulimit
 - sysctl.conf
 - net.unix.max_dgram_qlen = 100
 - net.core.wmem_max
 - net.ipv4.tcp_tw_recycle
 - 其他重要配置
 - 开启CoreDump
 - 查看配置是否生效

ulimit

设置系统打开文件数设置，解决高并发下 too many open files 问题。此选项直接影响单个进程容纳的客户端连接数。

Soft open files 是Linux系统参数，影响系统单个进程能够打开最大的文件句柄数量，这个值会影响到长连接应用如聊天中单个进程能够维持的用户连接数，运行ulimit -n能看到这个参数值，如果是1024，就是代表单个进程只能同时最多只能维持1024甚至更少（因为有其它文件的句柄被打开）。如果开启4个进程维持用户连接，那么整个应用能够同时维持的连接数不会超过4*1024个，也就是说最多只能支持4x1024个用户在线可以增大这个设置以便服务能够维持更多的TCP连接。

Soft open files 修改方法：

(1) ulimit -HSn 102400

这只是在当前终端有效，退出之后，open files 又变为默认值。

(2) 将ulimit -HSn 102400写到/etc/profile中，这样每次登录终端时，都会自动执行/etc/profile。

(3) 令修改open files的数值永久生效，则必须修改配置文件：/etc/security/limits.conf. 在这个文件后加上：

- soft nofile 1024000
- hard nofile 1024000
- root soft nofile 1024000
- root hard nofile 1024000

注意，修改limits.conf文件后，需要重启系统生效

sysctl.conf

打开文件 `/etc/sysctl.conf`，增加以下设置

```
#该参数设置系统的TIME_WAIT的数量，如果超过默认值则会被立即清除
net.ipv4.tcp_max_tw_buckets = 20000
#定义了系统中每一个端口最大的监听队列的长度，这是个全局的参数
net.core.somaxconn = 65535
#对于还未获得对方确认的连接请求，可保存在队列中的最大数目
net.ipv4.tcp_max_syn_backlog = 262144
#在每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许送到队列的数据包的最大数目
net.core.netdev_max_backlog = 30000
#能够更快地回收TIME-WAIT套接字。此选项会导致处于NAT网络的客户端超时，建议为0
net.ipv4.tcp_tw_recycle = 0
#系统所有进程一共可以打开的文件数量
fs.file-max = 6815744
#防火墙跟踪表的大小。注意：如果防火墙没开则会提示error: "net.netfilter.nf_conntrack_max" is
an unknown key, 忽略即可
net.netfilter.nf_conntrack_max = 2621440
```

运行 `sysctl -p`即可生效。

说明：

`/etc/sysctl.conf` 可设置的选项很多，其它选项可以根据自己的环境需要进行设置

net.unix.max_dgram_qlen = 100

swoole使用unix socket dgram来做进程间通信，如果请求量很大，需要调整此参数。系统默认为10，可以设置为100或者更大。

或者增加worker进程的数量，减少单个worker进程分配的请求量。

net.core.wmem_max

修改此参数增加socket缓存区的内存大小

```
net.ipv4.tcp_mem = 379008 505344 758016
net.ipv4.tcp_wmem = 4096 16384 4194304
net.ipv4.tcp_rmem = 4096 87380 4194304
net.core.wmem_default = 8388608
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_tw_reuse
```

是否socket reuse，此函数的作用是Server重启时可以快速重新使用监听的端口。如果没有设置此参数，

会导致server重启时发生端口未及时释放而启动失败

net.ipv4.tcp_tw_recycle

使用socket快速回收，短连接Server需要开启此参数。此参数表示开启TCP连接中TIME-WAIT sockets的快速回收，Linux系统中默认为0，表示关闭。打开此参数可能会造成NAT用户连接不稳定，请谨慎测试后再开启。

其他重要配置

- net.ipv4.tcp_syncookies=1
- net.ipv4.tcp_max_syn_backlog=81920
- net.ipv4.tcp_synack_retries=3
- net.ipv4.tcp_syn_retries=3
- net.ipv4.tcp_fin_timeout = 30
- net.ipv4.tcp_keepalive_time = 300
- net.ipv4.tcp_tw_reuse = 1
- net.ipv4.tcp_tw_recycle = 1
- net.ipv4.ip_local_port_range = 20000 65000
- net.ipv4.tcp_max_tw_buckets = 200000
- net.ipv4.route.max_size = 5242880

开启CoreDump

设置内核参数

```
kernel.core_pattern = /data/core_files/core-%e-%p-%t
```

通过ulimit -c命令查看当前coredump文件的限制

```
ulimit -c
```

如果为0，需要修改/etc/security/limits.conf，进行limit设置。

开启core-dump后，一旦程序发生异常，会将进程导出到文件。对于调查程序问题有很大的帮助

查看配置是否生效

如：修改net.unix.max_dgram_qlen = 100后，通过

```
cat /proc/sys/net/unix/max_dgram_qlen
```

如果修改成功，这里是新设置的值。

对象池

对象池

- [对象池](#)
 - [如何使用对象池](#)
 - [需要注意的](#)
 - [框架中的对象池](#)
 - [监控](#)

SD框架中重要的概念：对象池

框架中大量的使用了对象池模式，对象池模式可以很轻松的实现对象的复用，而不必频繁的GC和New，使得内存平滑减少泄露的风险。

那么如何使用对象池技术，以及需要注意什么呢。

如何使用对象池

框架提供了Pool类方便你使用对象池技术。

```
Pool::getInstance()->get(TaskCoroutine::class)->init($this->task_proxy_data, $dst_worker_id);
```

例如上面的例子，我们就获取了TaskCoroutine的一个实例。如果对象池中沒有TaskCoroutine那么对象池会new一个出来，使用完对象后需要归还给对象池。

```
Pool::getInstance()->push($taskCoroutine);
```

\$taskCoroutine为TaskCoroutine的一个实例。

需要注意的

对象池技术是复用对象，减少new的次数，那么就请注意__construct构造方法只会在第一次new的时候起作用，之后便不会再调用，所以建议大家书写对象池对象的时候用init作为初始化对象的方法并返回自身。

```
class TaskCoroutine extends CoroutineBase
{
    public function __construct()
    {
        parent::__construct();
    }
}
```

```
}

public function init($task_proxy_data, $id)
{
    $this->task_proxy_data = $task_proxy_data;
    $this->id = $id;
    $this->send(function ($serv, $task_id, $data) {
        $this->result = $data;
    });
    return $this;
}
}
```

每次从对象池中获取对象的时候调用init方法进行初始化。

框架中的对象池

框架中Controller，Model以及协成任务等都是使用了对象池技术，所以大家应该明白为什么尽量不要在__construct函数中书写自己的代码，因为并不是每次使用对象都会调用__construct，只会在new的使用调用__construct方法，也就是说在__construct中声明的参数在整个生命周期里都有效。

监控

提供了对象池的监控大家可以订阅'\$SYS/SD-1/status'来获取对象池的实时状态，SD-1是本机节点的名称。当然你还可以直接订阅 '\$SYS/#' 来获取所有监控。

VIP用户可以获取一个可视化的监控后台来实时观察对象池状态。

上下文-Context

上下文-Context

- [上下文-Context](#)
 - [Controller , Model , Task](#)
 - [RunStack](#)

middleware , controller , model , task , 在一个消息流程中共享同一个上下文Context。

其中middleware , controller , model可以修改Context , 而task只能读取不能修改。

例如MonitorMiddleware

```
public function after_handle($path)
{
    $this->context['path'] = $path;
    $this->context['execution_time'] = (microtime(true) - $this->start_run_time)
* 1000;
    if (self::$efficiency_monitor_enable) {
        $this->log('Monitor');
    }
}
```

这里将context附加了path和execution_time , 在after_handle流程前所有对context的操作都会影响到这里的context。

Controller , Model , Task

通过getContext获取当前的上下文。

RunStack

Context中包含RunStack字段 , 这个字段将记录消息流程中所执行的Middleware , Controller , Model , Task的方法。

例如 :

```
Array
(
    [RunStack] => Array
        (
            [0] => Server\Middlewares\MonitorMiddleware::before_handle
            [1] => Server\Middlewares\NormalHttpMiddleware::before_handle
```

```
[2] => Server\Controllers\TestController::setRequestResponse
[3] => Server\Controllers\TestController::http_testContext
[4] => Server\Models\TestModel::contextTest
[5] => Server\Tasks\TestTask::contextTest
[6] => Server\CoreBase\TaskProxy::startTask
[7] => Server\Models\TestModel::destroy
[8] => Server\Controllers\TestController::destroy
[9] => Server\Middlewares\NormalHttpMiddleware::after_handle
[10] => Server\Middlewares\MonitorMiddleware::after_handle
)

[request_id] => 15081258371921825039
[controller_name] => TestController
[method_name] => TestController:http_testContext
[test] => 1
[path] => /TestController/testContext
[execution_time] => 4.1890144348145
)
```

通过这个可以精确判断发生异常和错误的位置，也可以了解到SD框架的工作流程。

中间件

中间件

- 中间件
 - SD自带的中间件
 - Middleware
 - HttpMiddleware
 - NormalHttpMiddleware
 - MonitorMiddleware
 - 配置
 - 注意
 - interrupt
 - before
 - after
 - context

SD框架引入了中间件过程，消息的传递流程如下。

```
message->pack->middleware1(before)->middleware2(before)->...->route->controller->...->middleware2(after)->middleware1(after)
```

SD自带的中间件

Middleware

中间件最基础的类，开发中间件继承Middleware。

- before_handle 中间件before过程
- after_handle 中间件after过程
- interrupt 中断中间件

HttpMiddleware

http使用的基础中间件。一般开发http中间件请继承HttpMiddleware，提供了常用的命令。

NormalHttpMiddleware

这个中间件是提供给Http使用的，它具备的功能是提供了默认主页，404页面，文件后缀查询。

```
$config['ports'][] = [
```

```
'socket_type' => PortManager::SOCK_HTTP,
'socket_name' => '0.0.0.0',
'socket_port' => 8081,
'route_tool' => 'NormalRoute',
'middlewares' => ['MonitorMiddleware', 'NormalHttpMiddleware']
];
```

在ports配置中请务必携带NormalHttpMiddleware。

MonitorMiddleware

效率中间件，将记录接口的运行时间，并写入日志。

配置

中间件作用于每一个端口配置，也就是说不同的端口可以单独配置中间件。

中间件的执行严格按照数据的顺序执行。

```
$config['ports'][] = [
    'socket_type' => PortManager::SOCK_HTTP,
    'socket_name' => '0.0.0.0',
    'socket_port' => 8081,
    'route_tool' => 'NormalRoute',
    'middlewares' => ['MonitorMiddleware', 'NormalHttpMiddleware']
];
```

如上图执行顺序为MonitorMiddleware(before)->NormalHttpMiddleware(before)->...->NormalHttpMiddleware(after)->MonitorMiddleware(after).

注意

before流程中只要有一个中间件调用了interrupt方法，那么后续的中间件都不会被执行。

after流程中会忽略interrupt。

interrupt

一旦在before阶段触发了interrupt，那么将跳过Route，Controller的处理直接进入after的操作。

before

可以在after阶段修改信息流。

比如NormalHttpMiddleware中就修改了request中的值，导致后续的路由发生改变。

```
if (is_string($index)) {
```

```
$www_path = $this->getHostRoot($host) . $this->getHostIndex($host);
$result = httpEndFile($www_path, $this->request, $this->response);
if (!$result) {
    $this->redirect404();
} else {
    $this->interrupt();
}
} elseif (is_array($index)) {
    $this->request->server['path_info'] = "/" . implode("/", $index); //这里修改了request的值
} else {
    $this->redirect404();
}
```

after

after一般用于统计，比如MonitorMiddleware的作用就是统计各个接口的执行信息，并写入日志。

context

贯穿整个流程的context，是个上下文，context在整个流程中共享。任意一个地方修改context都会影响到整个流程。

可以通过context给后续流程传递信息。

进程管理

进程管理

[创建自定义进程](#)

[进程间RPC](#)

[自定义进程如何使用连接池](#)

SD可以创建自定义进程，可以处理很多复杂的业务。

注意

自定义进程是被保护的，kill掉或者挂掉会自动拉起。

自定义进程不受到reload的影响，需要自行kill重启。

自定义进程默认是异步的。

创建自定义进程

创建自定义进程

- [创建自定义进程](#)
 - [Process](#)
 - [start](#)
 - [onShutDown](#)
 - [设置进程RPC代理类](#)
 - [注解@oneWay](#)

Process

你需要继承Process类

```
class MyProcess extends Process
{
    public function start($process)
    {

    }

    public function getData()
    {
        return '123';
    }
}
```

你需要在APPServer中添加这个进程

```
/**
 * 用户进程
 */
public function startProcess()
{
    parent::startProcess();
    ProcessManager::getInstance()->addProcess(MyProcess::class);
}
```

addProcess拥有2个额外参数，\$name用于区分同种进程，\$params传递给进程的值，可以在进程中通过\$this->params获取。

```
public function addProcess($class_name, $name = '', $params = [])
```

addProcess允许你添加多个重复的进程，但每个进程需要不同的名称。

start

start是代表这个进程开始工作的回调方法，只会调用一次，用户可以在这里进行初始化操作。

onShutDown

代表进程结束的回调，伴随服务器的关闭，管理进程会逐一执行进程的onShutDown方法，这里可以做进程的收尾操作。

设置进程RPC代理类

进程间通讯是通过RPC进行的，会访问进程的public方法，用户可以设置RPC的代理类，这样RPC访问的将是代理类的方法。

```
public function __construct($name, $worker_id, $coroutine_need = true)
{
    parent::__construct($name, $worker_id, $coroutine_need);
    $this->setRPCProxy(new Test($this));
}
```

这样Test类将作为这个进程的RPC代理的类，Test类中public方法将被暴露出来。

AppServer继承的基类也是ProcessRPC，同样也可以设置setRPCProxy代理类。

注解@oneWay

代理类可以添加@oneWay注解，包含这个注解的方法进行RPC的时候将会被当做oneway处理。

```
/**
 * @param mixed $offset
 * @param mixed $value
 * @oneWay
 */
public function offsetSet($offset, $value)
{
    $this->map->offsetSet($offset, $value);
}
```

oneWay的意思是该方法不需要返回值

进程间RPC

进程间RPC

- [进程间RPC](#)
 - [与自定义进程的通讯](#)
 - [getRpcCall的函数原型](#)
 - [与worker进程的通讯](#)
 - [获取当前进程的ID](#)
 - [异常](#)

使用者可以通过ProcessManager实现worker进程与自定义进程间的RPC通讯。

与自定义进程的通讯

```
$data = ProcessManager::getInstance()->getRpcCall(SDHelpProcess::class)->getData(ConsoleHelp::DISPATCH_KEY);
```

与SDHelpProcess进程的一次RPC通讯。

getData是SDHelpProcess类的一个public方法。

getRpcCall的函数原型

```
function getRpcCall($class_name, $oneWay = 'auto', $name = '')
```

oneWay如果为true意思为这是一次单向通讯。

oneWay为auto代表会通过注解判断该方法是否是单向通讯，如果没有注解默认false。

注解只支持RPC代理类

与worker进程的通讯

```
$result = ProcessManager::getInstance()->getRpcCallWorker(0)->getPoolStatus();
```

这是一次与workerId = 0 的进程通讯。

getPoolStatus是AppServer或者其基类中的public方法。

这个函数也可以与自定义进程通讯，前提是你知道那个自定义进程的workerId。

获取当前进程的ID

通过下面命令可以获取当前进程的ID

```
get_instance()->getWorkerId();
```

异常

RPC中需要返回个异常直接通过Throw方法即可，调用方会获得这个异常。

自定义进程如何使用连接池

自定义进程如何使用连接池

我们需要在start方法中创建连接池。

方法和AppServer中initAsynPools类似。

```
class MyProcess extends Process
{
    protected $redisPool;
    protected $mysqlPool;
    public function start($process)
    {
        $this->redisPool = new RedisAsynPool($this->config, $this->config->get('redis.active'));
        $this->mysqlPool = new MysqlAsynPool($this->config, $this->config->get('mysql.active'));
        get_instance()->addAsynPool("redisPool",$this->redisPool);
        get_instance()->addAsynPool("mysqlPool",$this->mysqlPool);
    }
}
```

get_instance()->addAsynPool只是为了增加兼容性，其实可以不需要。

异步连接池

异步连接池

Redis

Redis

- [Redis](#)
 - [创建Redis连接池](#)
 - [获取Redis连接池](#)
 - [使用方法](#)
 - [Redis-LUA](#)
 - [同步Redis](#)

SD提供了Redis异步连接池，连接池的使用和正常Redis扩展一样。

创建Redis连接池

在SD的initAsynPools方法中已经创建好了redis和mysql默认的连接池。

```
/**
 * 初始化各种连接池
 * @param $workerId
 */
public function initAsynPools($workerId)
{
    $this->asynPools = [];
    if ($this->config->get('redis.enable', true)) {
        $this->asynPools['redisPool'] = new RedisAsynPool($this->config, $this->
config->get('redis.active'));
    }
    if ($this->config->get('mysql.enable', true)) {
        $this->asynPools['mysqlPool'] = new MysqlAsynPool($this->config, $this->
config->get('mysql.active'));
    }
    $this->redis_pool = $this->asynPools['redisPool'] ?? null;
    $this->mysql_pool = $this->asynPools['mysqlPool'] ?? null;
}
```

如果你想创建多个Redis连接池可以仿照上面的方法。

```
$this->addAsynPool('redisPool2', new RedisAsynPool($this->config, 'redis2'));
```

获取Redis连接池

在Controller,Model,Task共同的基类CoreBase中默认获取了RedisPool。

```

/**
 * 当被loader时会调用这个方法进行初始化
 * @param $context
 */
public function initialization(&$context)
{
    $this->setContext($context);
    $this->redis = $this->loader->redis("redisPool");
    $this->db = $this->loader->mysql("mysqlPool",$this);
}

```

使用方法

Redis的使用方法和PhpRedis扩展一致，可以参考ServerRedisTest，这是关于Redis的测试用例。

```
$value = $this->redis->set('test', 'testRedis');
```

Redis-LUA

SD支持Redis-LUA，根目录下有个名为lua的文件夹，这里的lua脚本都会自动被SD加载进Redis中。

```

public function http_testRedisLua()
{
    $value = $this->redis->evalSha(getLuaSha1('sadd_from_count'), ['testlua', 100],
2, [1, 2, 3]);
    $this->http_output->end($value);
}

```

通过evalSha和getLuaSha1方法配合我们可以非常容易的使用Redis-LUA功能。其中sadd_from_count是lua文件夹某一个lua脚本的文件名。

在SD启动时我们也能看到加载了哪些lua脚本，如果redis服务器不支持lua也会有相应的提示。

```

! [NOTE] Press Ctrl-C to quit. Start success.

> [RLUA] 已加载sadd_from_count脚本
> [SYS] 已开启代码热重载
> [CatCache] 已完成加载缓存文件
^C

```

同步Redis

```
$redisSync = $this->redis_pool->getSync();
```


通过getSync返回一个同步的redis连接。

Mysql

Mysql

- Mysql
 - 迁移指南
 - 创建Mysql连接池
 - 获取Mysql连接池
 - 返回结构
 - 直接执行sql
 - Select
 - Update
 - Replace
 - Delete
 - Raw混合SQL
 - 获取SQL命令
 - insertInto,updateInto,replaceInto 批量操作
 - 超时
 - 事务
 - 同步Mysql

SD提供了Mysql异步连接池。这里提供3.1版本全新的API，新版Mysql支持SQL预处理请求，不用再担心安全问题了。

迁移指南

之前的版本迁移到新版时需要注意：

1. coroutineSend已经被弃用，更换为query。
2. 事务已改版

```
public function query($sql = null, callable $set = null)
```

创建Mysql连接池

在SD的initAsynPools方法中已经创建好了redis和mysql默认的连接池。

```
/**
 * 初始化各种连接池
 * @param $workerId
```

```

    */
    public function initAsynPools($workerId)
    {
        $this->asynPools = [];
        if ($this->config->get('redis.enable', true)) {
            $this->asynPools['redisPool'] = new RedisAsynPool($this->config, $this->config->get('redis.active'));
        }
        if ($this->config->get('mysql.enable', true)) {
            $this->asynPools['mysqlPool'] = new MysqlAsynPool($this->config, $this->config->get('mysql.active'));
        }
        $this->redis_pool = $this->asynPools['redisPool'] ?? null;
        $this->mysql_pool = $this->asynPools['mysqlPool'] ?? null;
    }

```

如果你想创建多个Mysql连接池可以仿照上面的方法。

```

$this->addAsynPool('mysqlPool2', new MysqlAsynPool($this->config, 'mysql2'));

```

获取Mysql连接池

在Controller,Model,Task中默认获取了MysqlPool。

```

/**
 * 当被loader时会调用这个方法进行初始化
 * @param $context
 */
public function initialization(&$context)
{
    $this->setContext($context);
    $this->redis = $this->loader->redis("redisPool");
    $this->db = $this->loader->mysql("mysqlPool",$this);
}

```

返回结构

通过query默认会返回一个数组。

```

{
    "result": [
    ],
    "affected_rows": 0,
    "insert_id": 0,
    "client_id": 0
}

```

result是返回的结果必定是个数组，affected_rows影响的行数，insert_id插入的id。

直接执行sql

```
$this->db->query("
    CREATE TABLE IF NOT EXISTS `MysqlTest` (
        `peopleid` smallint(6) NOT NULL AUTO_INCREMENT,
        `firstname` char(50) NOT NULL,
        `lastname` char(50) NOT NULL,
        `age` smallint(6) NOT NULL,
        `townid` smallint(6) NOT NULL,
        PRIMARY KEY (`peopleid`),
        UNIQUE KEY `unique_fname_lname` (`firstname`, `lastname`),
        KEY `fname_lname_age` (`firstname`, `lastname`, `age`)
    );
```

Select

```
$value = $this->db->Select('*')
    ->from('MysqlTest')
    ->where('townid', 10000)->query();
```

Update

```
$value = $this->db->update('MysqlTest')
    ->set('age', '20')
    ->where('townid', 10000)->query();
```

Replace

```
$value = $this->mysql_pool->dbQueryBuilder->replace('MysqlTest')
    ->set('firstname', 'White')
    ->set('lastname', 'Cat')
    ->set('age', '26')
    ->set('townid', '10000')->query();
```

Delete

```
$value = $this->mysql_pool->dbQueryBuilder->delete()
    ->from('MysqlTest')
    ->where('townid', 10000)->query();
```

Raw混合SQL

```
$selectMiner = $this->db->select('*')->from('account');
$selectMiner = $selectMiner->where('', '(status = 1 and dec in ("ss", "cc")) or name = "kk"', Miner::LOGICAL_RAW)->query();
```

获取SQL命令

```
$value = $this->db->insertInto('account')->intoColumns(['uid', 'static'])->intoValues([[36, 0], [37, 0]])->getStatement(true);
```

通过getStatement可以获取构建器构建的sql语法。

insertInto,updateInto,replaceInto 批量操作

```
$value = $this->db->insertInto('account')->intoColumns(['uid', 'static'])->intoValues([[36, 0], [37, 0]])->query();
```

批量命令配合intoColumns和intoValues设置批量操作

超时

超时设置

```
$result = $this->db->select('*')->from('account')->limit(1)->query(null,function (MySQLCoroutine $mySqlCoroutine){
    $mySqlCoroutine->setTimeout(1000);
    $mySqlCoroutine->noException("test");
    $mySqlCoroutine->dump();
});
echo $result->num_rows();
```

除了超时Mysql提供了一些其他的API

- result_array 直接返回result
- row_array 返回某一个
- row 返回第一个
- num_rows 返回数量
- insert_id 返回insert_id
- dump 打印执行的sql命令

事务

begin开启一个事务，在回调中执行事务，如果抛出异常则自动回滚，如果正常则自动提交。

```
public function http_mysql_begin_coroutine_test()
{
    $this->db->begin(function ()
    {
        $result = $this->db->select("*")->from("account")->query();
        var_dump($result['client_id']);
        $result = $this->db->select("*")->from("account")->query();
        var_dump($result['client_id']);
    });
    $this->http_output->end(1);
}
```

同步Mysql

```
$mysqlSync = $this->mysql_pool->getSync();
```

通过getSync返回一个同步PDO的mysql连接。

同步mysql可以使用pdo开头的方法。

- pdoQuery
- pdoBeginTrans
- pdoCommitTrans
- pdoRollBackTrans
- pdoInsertId

Mqtt

Mqtt

- [Mqtt](#)
 - [创建MQTT连接](#)
 - [回调](#)
 - [Api](#)

SD提供了MQTT异步客户端。

- 异步MQTT客户端，可以结合EMQ开源服务器实现百万千万级通讯业务。
- 支持完整的MQTT协议规则
- 纯异步，断线重连

创建MQTT连接

```
public function initAsynPools($workerId)
{
    parent::initAsynPools();
    $mqtt = new MQTT('tcp://127.0.0.1:11883/', 'root1');
    //设置持久会话
    $mqtt->setConnectClean(false);
    //认证
    $mqtt->setAuth('root1', 'root');
    //存活时间
    $mqtt->setKeepalive(3600);
    //回调
    $mqtt->on('publish', function ($mqtt, PUBLISH $publish_object) {
        printf(
            "\e[32mI got a message\e[0m:(msgid=%d, QoS=%d, dup=%d, topic=%s) \e[32m%
s\e[0m\n",
            $publish_object->getMsgID(),
            $publish_object->getQos(),
            $publish_object->getDup(),
            $publish_object->getTopic(),
            $publish_object->getMessage()
        );
    });
    $mqtt->on('connack', function (MQTT $mqtt, CONNACK $connack_object) {
        var_dump("MQTT连接成功");
        $topics['$SYS/#'] = 1;
        $mqtt->subscribe($topics);
    });
    $mqtt->connect();
}
```

我们在initAsynPools中创建了Mqtt客户端，这样会导致每个worker进程都会创建一个MQTT客户端。

最好的做法是创建一个自定义进程，在自定义进程中创建MQTT客户端。

MQTT是纯异步的，需要通过回调获取通讯的结果。

回调

- connack 连接成功的回调
- disconnect 断开连接的回调
- puback
- pubrec
- pubrel
- pubcomp
- suback 订阅成功的回调
- unsuback 移除订阅的回调
- publish 收到订阅消息的回调

Api

- setRetryTimeout 设置Retry超时时间
- setVersion 设置MQTT版本
- version 获取MQTT版本
- setAuth 设置AUTH
- setKeepalive 设置Keepalive时间
- setConnectClean 设置连接清除标志
- setWill 设置遗嘱消息
- on 设置回调
- ping ping
- connect 连接
- disconnect 断开连接
- publish 发布
- subscribe 订阅
- unsubscribe 取消订阅

HttpClient

HttpClient

Http请求客户端连接池。

- [HttpClient](#)
 - [添加连接池](#)
 - [使用方法](#)

使用方式简单，步骤按照下面的来。

添加连接池

首先我们打开AppServer，添加连接池的声明。

```
public function initAsynPools()
{
    parent::initAsynPools();
    $this->addAsynPool('GetIPAddress', new HttpClientPool($this->config, 'http://i
nt.dpool.sina.com.cn'));
    $this->addAsynPool('WeiXinAPI', new HttpClientPool($this->config, 'https://api
.weixin.qq.com'));
}
```

我们通过继承initAsynPools的方法为我们的框架添加了2个http连接池。

HttpClientPool的构造函数中\$config传框架的Config实例，\$baseUrl请注意如果你要访问https://api.weixin.qq.com/1/2/34/5/6/abc 类似这样的网址，那么它的baseUrl为https://api.weixin.qq.com。

使用方法

我们打开我们需要进行访问的代码，以下我们假设在controller中进行访问。

```
/**
 * @var HttpClientPool
 */
protected $GetIPAddressHttpClient;
public function initialization($controller_name, $method_name)
{
    parent::initialization($controller_name, $method_name);
    $this->GetIPAddressHttpClient = get_instance()->getAsynPool('GetIPAddress');
}
```

我们在initialization初始化函数中获得这个HttpClientPool。

连接池可以在__construct和initialization中初始化，区别在于initialization每次访问都会执行，连接池只需要初始化一次即可，推荐在__construct进行初始化。

接下来我们就可以在代码中书写我们需要访问的api了。

```
public function http_ip_test()  
{  
    $ip = $this->http_input->server('remote_addr');  
    $response = $this->GetIPAddressHttpClient->httpClient  
        ->setQuery(['format' => 'json', 'ip' => $ip])  
        ->coroutineExecute('/iplookup/iplookup.php');  
}
```

Client

Client

TCP客户端连接池

- [Client](#)
 - [添加连接池](#)
 - [使用方法](#)

添加连接池

首先我们打开AppServer，添加连接池的声明。

```
$this->addAsynPool('RPC',new TcpClientPool($this->config,'test',"192.168.0.1:9093"))
;
```

test指的是应用的配置名

使用tcpClientPool需要注意的是，你需要配置client.php

```
$config['tcpClient']['asyn_max_count'] = 10;
$config['tcpClient']['test']['pack_tool'] = 'JsonPack';
```

- asyn_max_count指的是连接池最大的数量
- pack_tool是使用封装器的名称。

注意，如果是和SD服务器通讯那么请使用SdTcpRpcPool，而不是TcpClientPool。

使用方法

首先获取连接池

```
/**
 * @var TcpClientPool
 */
public $rpc;
public function __construct($proxy = ChildProxy::class)
{
    parent::__construct($proxy);
    $this->rpc = get_instance()->getAsynPool("RPC");
}
```

然后我们开始写业务逻辑

```
public function http_tcpClient()
{
    $data = ['controller_name'=>"TestController", "method_name"=>"testTcp", "data"=>"test"];
    $this->rpc->setPath("TestController/testTcp", $data);
    $result = $this->rpc->coroutineSend($data);
    $this->http_output->end($result);
}
```

这里setPath方法将为\$data添加一个path字段，框架规定必须有path字段，当然这个path字段不会被作为消息发出。

其中controller_name和method_name是作为SD的NormalRoute路由必须条件存在的字段，这个请根据需要访问的服务器实际情况进行设置。

框架会根据设置的封装器进行消息的封装，coroutineSend后面只需要携带原始消息，不必做多余的封装。

封装协议的操作请到封装器中书写。

AMQP

AMQP

通过php-amqplib/php-amqplib库和eventloop实现异步回调。

底层是基于stream库并不是swoole_client，write的操作依旧是堵塞的，不过这个影响已经极小，read的操作通过eventloop实现了异步读，所以可以实现异步的订阅。

比较具体的用法可以参考[php-amqplib](#)中的demo。但是有些细节需要注意，有所不同。

初始化的地点依旧是在AppServer的initAsynPools下,同样的道理建议使用自定义进程。

例子：

```
/**
 * 这里可以进行额外的异步连接池，比如另一组redis/mysql连接
 * @param $workerId
 * @return array
 */
public function initAsynPools($workerId)
{
    parent::initAsynPools($workerId);
    if($workerId==0) {
        $amqp = new AMQP('localhost',5672,'guest','guest');
        $channel = $amqp->channel();
        $channel->queue_declare('msgs', false, true, false, false);
        $channel->exchange_declare('router', 'direct', false, true, false);
        $channel->queue_bind('msgs', 'router');
        $channel->basic_consume('msgs', 'consumer', false, false, false, false, function (AMQPMessage $message)
        {
            echo "\n-----\n";
            echo $message->body;
            $message->delivery_info['channel']->basic_ack($message->delivery_info['delivery_tag']);
        });
    }
}
```

这里只在第一个进程中启动了AMQP客户端。

异步AMQP客户端是由Server\Asyn\AMQP\AMQP构建的，不要使用错了。

和php-amqplib库中demo不同的是，千万不要使用wait方法进行堵塞！

RPC

RPC

框架为开发者提供了RPC服务。

- [RPC](#)
 - [添加连接池](#)
 - [使用方法](#)
 - [原理](#)

RPC与TcpClientPool不同，RPC并没有连接池，是采用单个连接进行通讯的，比TcpClientPool更加省连接，但是需要更完善的规范，目前RPC只适用于SD服务器之间的通讯。

添加连接池

```
$this->addAsynPool('RPC',new SdTcpRpcPool($this->config,'test',"192.168.0.1:9093"));
```

使用方法

用法和TcpClientPool类似但是为了方便增加了个帮助函数。

```
function helpToBuildSDControllerQuest($context, $controllerName, $method)
```

其中\$context就是Controller，Model中的context。通过helpToBuildSDControllerQuest获取构建后的data，再通过coroutineSend发出请求，可以加上oneway标示表明这是一个单向请求。

```
public function http_tcp()  
{  
    $this->sdrpc = get_instance()->getAsynPool('RPC');  
    $data = $this->sdrpc->helpToBuildSDControllerQuest($this->context, 'MathService'  
, 'add');  
    $data['params'] = [1, 2];  
    $result = $this->sdrpc->coroutineSend($data);  
    $this->http_output->end($result);  
}
```

原理

SdTcpRpcPool使用的是单个连接进行RPC操作，每次请求都会携带一个token字段，发送的token和接收

token是保持一致的，token会自增加。

日志工具-GrayLog

日志工具-GrayLog

- [日志工具-GrayLog](#)
 - [Docker部署](#)
 - [Graylog设置Input](#)

SD框架的日志接入了graylog，默认还是file模式，可以在config/log.php中改变配置。

你需要自己部署graylog，或者直接通过docker部署。

[这里是graylog的文档](#)

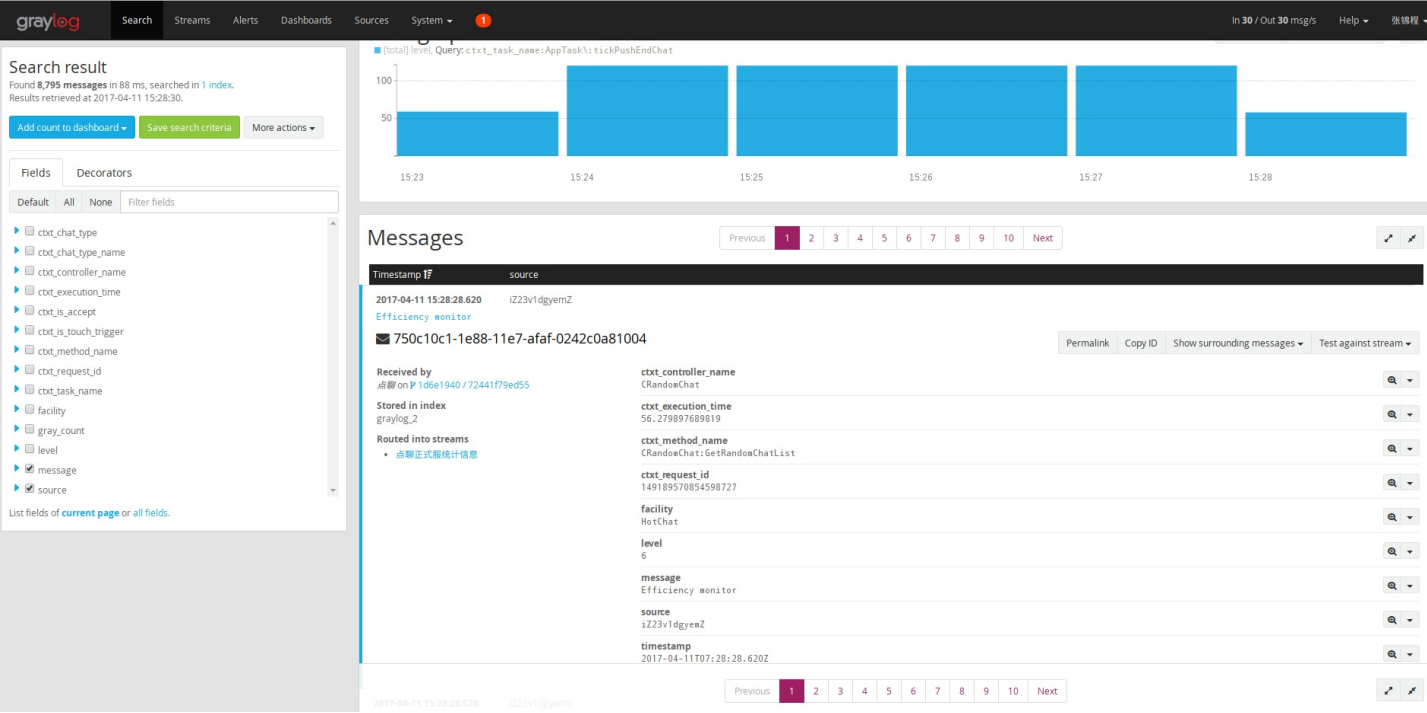
```
$config['log']['active'] = 'graylog';
$config['log']['log_level'] = \Monolog\Logger::DEBUG;
$config['log']['log_name'] = 'SD';

$config['log']['graylog']['udp_send_port'] = 12500;
$config['log']['graylog']['ip'] = '127.0.0.1';
$config['log']['graylog']['port'] = '12201';
$config['log']['graylog']['api_port'] = '9000';
$config['log']['graylog']['efficiency_monitor_enable'] = true;

$config['log']['file']['log_path'] = '/../../';
$config['log']['file']['log_max_files'] = 15;
$config['log']['file']['efficiency_monitor_enable'] = false;
```

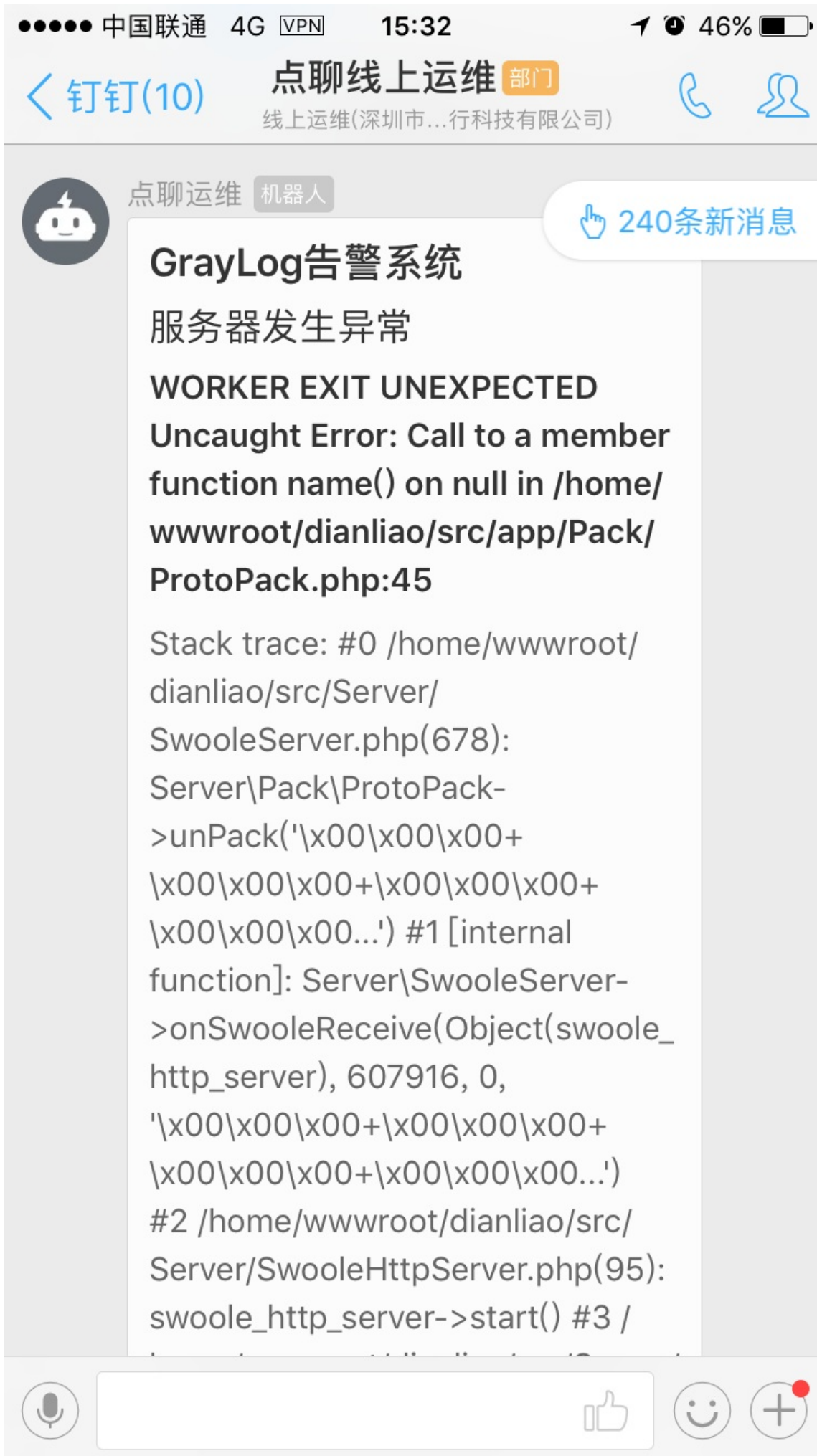
udp_send_port是框架开启的udp发送端口，port是graylog的udp接收端口，api_port是graylog提供的api访问端口，graylog默认是开始efficiency_monitor_enable，这个会对sd的访问进行效率监控。

默认访问grayloag是http://localhost:9000。



此外还可以通过graylog做更多的事，比如数据统计，数据分析，结合钉钉机器人做告警系统。

比如搭建完成的例子，一旦服务器出现异常，运维群就会收到信息，而这一切都是由graylog服务器分析日志后调用的和本身的业务服务器没有关系：



Docker部署

这里提供一份docker-compose的配置文件，请注意日志所需要的磁盘空间非常的大，这里我们挂载了一个磁盘到了/data目录，将文件存盘到/data目录对应的文件夹下。

```
version: '2'
services:
  # MongoDB: https://hub.docker.com/_/mongo/
  mongodb:
    image: mongo:3
    volumes:
      - /data/docker/mongodb:/data/db
  # Elasticsearch: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/docker.html
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:5.6.3
    volumes:
      - /data/docker/elasticsearch:/usr/share/elasticsearch/data
    environment:
      - http.host=0.0.0.0
      - transport.host=localhost
      - network.host=0.0.0.0
      # Disable X-Pack security: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/security-settings.html#general-security-settings
      - xpack.security.enabled=false
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
      mem_limit: 1g
  # Graylog: https://hub.docker.com/r/graylog/graylog/
  graylog:
    image: graylog/graylog:2.4.0-1
    volumes:
      - /data/docker/graylog:/usr/share/graylog/data/journal
    environment:
      # CHANGE ME!
      - GRAYLOG_PASSWORD_SECRET=somepasswordpepper
      # Password: admin
      - GRAYLOG_ROOT_PASSWORD_SHA2=8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918
      - GRAYLOG_WEB_ENDPOINT_URI=http://114.55.253.83:9000/api
    links:
      - mongodb:mongo
      - elasticsearch
    depends_on:
      - mongodb
      - elasticsearch
    ports:
```

```
# Graylog web interface and REST API
- 9000:9000
# Syslog TCP
- 514:514
# Syslog UDP
- 514:514/udp
# GELF TCP
- 12201:12201
# GELF UDP
- 12201:12201/udp
```

Graylog设置Input

启动好Graylog后，需要配置Input，这里注意我们需要选择GelfUdp默认是12201端口。如果一切顺利，防火墙ok的话就能接收到SD发来的日志了。

微服务-Consul

微服务-Consul

如果想完全体验微服务框架，那么还需安装consul。

[这里是consul的文档](#)

可以直接下载consul的二进制文件进行搭建consul服务器集群，或者通过docker部署。

不搭建好consul服务端是没法使用微服务的哦，consul-agent框架会自动启动，开发者需要手动搭建consul-server。

Consul基础

Consul基础

Consul是google开源的一个使用go语言开发的服务发现、配置管理中心服务。内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value存储、多数据中心方案，不再需要依赖其他工具（比如ZooKeeper等）。服务部署简单，只有一个可运行的二进制的包。每个节点都需要运行agent，他有两种运行模式server和client。每个数据中心官方建议需要3或5个server节点以保证数据安全，同时保证server-leader的选举能够正确的进行。

client

CLIENT表示consul的client模式，就是客户端模式。是consul节点的一种模式，这种模式下，所有注册到当前节点的服务会被转发到SERVER，本身是不持久化这些信息。

SD框架会自动解析配置启动client模式。

server

SERVER表示consul的server模式，表明这个consul是个server，这种模式下，功能和CLIENT都一样，唯一不同的是，它会把所有的信息持久化的本地，这样遇到故障，信息是可以被保留的。

需要用户自己部署

server-leader

中间那个SERVER下面有LEADER的字眼，表明这个SERVER是它们的老大，它和其它SERVER不一样的一点是，它需要负责同步注册的信息给其它的SERVER，同时也要负责各个节点的健康监测。

多个server保障不会出现单点故障，测试中可以只开启一个consul-server，最佳是3个。

raft

server节点之间的数据一致性保证，一致性协议使用的是raft，而zookeeper用的paxos，etcd采用的也是raft。

服务发现协议

consul采用http和dns协议，etcd只支持http

SD使用的是http协议和consul通讯

服务注册

consul支持两种方式实现服务注册，一种是通过consul的服务注册http API，由服务自己调用API实现注册，另一种方式是通过json个是的配置文件实现注册，将需要注册的服务以json格式的配置文件给出。consul官方建议使用第二种方式。

SD使用的是json配置文件，但无需复杂的配置，json文件会自动生成

服务发现

consul支持两种方式实现服务发现，一种是通过HTTP API来查询有哪些服务，另外一种是通过consul agent 自带的DNS（8600端口），域名是以NAME.service.consul的形式给出，NAME即在定义的服务配置文件中，服务的名称。DNS方式可以通过check的方式检查服务。

SD使用HTTP API查询服务

服务间的通信协议

Consul使用gossip协议管理成员关系、广播消息到整个集群，他有两个gossip pool（LAN pool和WAN pool），LAN pool是同一个数据中心内部通信的，WAN pool是多个数据中心通信的，LAN pool有多个，WAN pool只有一个。

搭建Consul服务器

搭建consul服务器

想使用微服务第一件事就是搭建好consul服务器。

consul服务器请选用独立的物理机，配置无需太好，也可以用docker。

安装consul

为了安装Consul，需要在[下载页面](#)中找到和你系统匹配的包。Consul被打包成zip格式的压缩包。

解压Consul zip包，复制consul二进制文件到系统PATH中包含的路径下，以确保它可以被执行。在Unix系统中，~/bin和/usr/local/bin是通常的安装路径，选择哪个依赖于你安装Consul给单个用户使用还是所有用户都可以使用。

验证安装

安装Consul后，通过打开新的终端回话并且输入consul是否可用来验证安装是否工作。通过执行consul你应该可以看到下面类似的输出：

```
$ consul
usage: consul [--version] [--help] <command> [<args>]

Available commands are:
  agent      Runs a Consul agent
  event      Fire a new event
  exec       Executes a command on Consul nodes
  force-leave Forces a member of the cluster to enter the "left" state
  info       Provides debugging information for operators
  join       Tell Consul agent to join cluster
  keygen     Generates a new encryption key
  leave      Gracefully leaves the Consul cluster and shuts down
  members    Lists the members of a Consul cluster
  monitor    Stream logs from a Consul agent
  reload     Triggers the agent to reload configuration files
  version    Prints the Consul version
  watch      Watch for changes in Consul
```

如果终端报告consul没有被找到的错误，那可能是你的PATH没有被正确的设置导致的。请回到前面的步骤去检查你的PATH环境变量是否包含了安装Consul目录。

启动

在Consul安装完成后，必须先运行代理。该代理可以以服务器或者客户端模式运行。每个数据中心必须至少包含一个服务器，不过一个集群推荐3或5个服务器。一个单服务器的部署在失败的情况下会发生数据丢失因此不推荐使用。

所有其他的代理运行在客户端模式。一个客户端是一个非常轻量级的进程，它注册服务，运行健康检查，以及转发查询到服务器。代理必须运行在集群中的每个节点上。

客户端模式的代理SD框架会自动创建

server的搭建过程，这里我们采用了3个服务器进行搭建。

cn1：

```
#consul agent -bootstrap-expect 2 -server -data-dir /data/consul -node=cn1 -bind=192.168.1.202 -datacenter=dc1
```

cn2:

```
#consul agent -server -data-dir /data/consul -node=cn2 -bind=192.168.1.201 -datacenter=dc1 -join 192.168.1.202
```

cn3:

```
#consul agent -server -data-dir /data/consul -node=cn3 -bind=192.168.1.200 -datacenter=dc1 -join 192.168.1.202
```

参数解释：

- -bootstrap-expect:集群期望的节点数，只有节点数量达到这个值才会选举leader。
- -server：运行在server模式
- -data-dir：指定数据目录，其他的节点对于这个目录必须有读的权限
- -node：指定节点的名称
- -bind：为该节点绑定一个地址
- -datacenter: 数据中心名称，
- -join：加入到已有的集群中

如果没有那么多机器可以设置-bootstrap-expect 1 这样只启动一个consul-server节点也是可以的

SD中Consul配置

SD中Consul配置

SD框架需要部署consul client agent，这里就比较容易只需要将consul的二进制文件复制到bin/exec目录下命名为consul即可，然后在config/consul.php中进行相关配置。

```
//是否启用consul
$config['consul']['enable'] = true;
//数据中心配置
$config['consul']['datacenter'] = 'dc1';
//开放给本地
$config['consul']['client_addr'] = '127.0.0.1';
//服务器名称，同种服务应该设置同样的名称，用于leader选举
$config['consul']['leader_service_name'] = 'Test';
//node的名字，每一个都必须不一样，也可以为空自动填充主机名
$config['consul']['node_name'] = 'SD-1';
//consul的data_dir默认放在临时文件下
$config['consul']['data_dir'] = "/tmp/consul";
//consul join地址，可以是集群的任何一个，或者多个
$config['consul']['start_join'] = ["192.168.8.85"];
//本地网卡设备
$config['consul']['bind_net_dev'] = "enp2s0";
//监控服务
$config['consul']['watches'] = ['MathService', 'TestController'];
//发布服务
$config['consul']['services'] = ['MathService:9091', 'MathService:8081', 'TestController:9091', 'TestController:8081'];
//是否开启TCP集群，启动consul才有用
$config['cluster']['enable'] = true;
//TCP集群端口
$config['cluster']['port'] = 9999;

//***断路器设置***
//阈值
$config['fuse']['threshold'] = 0.01;
//检查时间
$config['fuse']['checktime'] = 2000;
//尝试打开的间隔
$config['fuse']['trytime'] = 1000;
//尝试多少个
$config['fuse']['trymax'] = 3;
```

- start_join 表示需要加入的consul集群ip地址
- bind_addr 填写本地网卡ip地址或者通过 bind_net_dev填写绑定网卡的设备名
- watches表示想监控的微服务controller名称
- services表示需要发布的微服务controller名称，需要写上端口

这样配置好之后，启动依赖的服务器集群，你可以通过consul提供的web界面查看服务的运行情况。

启动成功的打印：


```
> [RLUA] 已加载sadd_from_count脚本
> [SYS] 已开启代码热重载
> [CatCache] 已完成加载缓存文件
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
    Version: 'v0.7.5'
    Node ID: 'a6395f3d-cee1-4308-b3a6-b3433b54a189'
    Node name: 'SD-1'
    Datacenter: 'dc1'
    Server: false (bootstrap: false)
    Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
    Cluster Addr: 192.168.8.57 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    Atlas: <disabled>

==> Log data will now stream in as it occurs:

2018/03/05 14:58:48 [INFO] serf: EventMemberJoin: SD-1 192.168.8.57
2018/03/05 14:58:48 [INFO] agent: (LAN) joining: [192.168.8.85]
2018/03/05 14:58:48 [INFO] serf: EventMemberJoin: localhost.localdomain 192.168.
8.85
2018/03/05 14:58:48 [INFO] agent: (LAN) joined: 1 Err: <nil>
2018/03/05 14:58:48 [INFO] consul: adding server localhost.localdomain (Addr: tc
p/192.168.8.85:8300) (DC: dc1)
2018/03/05 14:58:48 [INFO] agent: Synced service 'Tcp_MathService'
2018/03/05 14:58:48 [INFO] agent: Synced service 'Http_MathService'
2018/03/05 14:58:48 [INFO] agent: Synced service 'Tcp_TestController'
2018/03/05 14:58:48 [INFO] agent: Synced service 'Http_TestController'
2018/03/05 14:58:48 [INFO] agent: Synced check 'service:Tcp_MathService'
> [CONSUL] 发现MathService(192.168.8.57:9091) TCP服务，应用配置consul_MathService
> [CONSUL] Leader变更，被选举为Leader
    2018/03/05 14:58:55 [INFO] agent: Synced check 'service:Tcp_TestController'
> [CONSUL] 发现TestController(192.168.8.57:9091) TCP服务，应用配置consul
    2018/03/05 14:58:56 [INFO] agent: Synced check 'service:Http_TestController'
> [CONSUL] 发现TestController(192.168.8.57:8081) HTTP服务
    2018/03/05 14:58:57 [INFO] agent: Synced check 'service:Http_MathService'
> [CONSUL] 发现MathService(192.168.8.57:8081) HTTP服务
```

[这里是consul web ui的介绍](#)

默认是<http://localhost:8500/ui>。




SERVICES

NODES

KEY/VALUE

ACL

DC1 ▾



Filter by name

any status ▾

EXPAND

MathService4 passing

consul1 passing

MathService

TAGS
http, tcp

NODES

SD-2192.168.8.482 passing

Serf Health StatusserfHealthpassing

Service 'MathService' check service:http_MathServicepassing

SD-2192.168.8.482 passing

Serf Health StatusserfHealthpassing

Service 'MathService' check service:Tcp_MathServicepassing

微服务

微服务

微服务分为RPC和REST方式。

REST

通过ConsulServices::getInstance()->getRESTService获取对应的服务。

```
public function http_testConsul()
{
    $rest = ConsulServices::getInstance()->getRESTService('MathService', $this->context);
    $rest->setQuery(['one' => 1, 'two' => 2]);
    $result = $rest->add();
    $this->http_output->end($result['body']);
}
```

RPC

通过ConsulServices::getInstance()->getRPCService获取对应的服务。

```
public function http_testConsul2()
{
    $rest = ConsulServices::getInstance()->getRPCService('MathService', $this->context);
    $result = $rest->add(1, 2);
    $this->http_output->end($result);
}

public function http_testConsul3()
{
    $rest = ConsulServices::getInstance()->getRPCService('MathService', $this->context);
    $result = $rest->call('add', [1, 2], false);
    $this->http_output->end($result);
}
```

call方法可以设置oneway模式，一旦使用oneway模式将不能接收到返回结果。

选举-Leader

Leader

配置中存在一个leader_service_name配置。

```
$config['consul']['leader_service_name'] = 'Test';
```

这个配置用于相同服务器选举用，使用相同名称的服务器会进选举，选出一个Leader。

```
Start::isLeader()
```

通过上面的方法可以获取到当前服务器是否为Leader。

定时任务中可以使用Start::isLeader()判断用于在Leader上执行定时任务。

Consul动态配置定时任务

Consul动态配置定时任务

可以通过Consul的KV存储动态配置定时任务

储存目录为TimerTask/{leader_service_name}/{task_name}

- leader_service_name consul配置中设置
- task_name 你命名的定时任务名称

存储的格式是Json，内容和TimerTask中的定义一样。

```
$config['timerTask'][] = [  
    //'start_time' => 'Y-m-d 19:00:00',  
    //'end_time' => 'Y-m-d 20:00:00',  
    'task_name' => 'TestTask',  
    'method_name' => 'test',  
    'interval_time' => '1',  
];
```

将上面内容转为Json即可。

通过timerTask配置设置的定时任务是没法通过consul修改的。
consul修改会立即影响定时任务。

熔断与降级

熔断与降级

请看一个例子

```
public function http_testConsul3()
{
    $rest = ConsulServices::getInstance()->getRPCService('MathService', $this->context);
    $result = $rest->call('sum', [10000000], false, function (TcpClientRequestCoroutine $clientRequestCoroutine){
        $clientRequestCoroutine->setTimeout(1000);
        $clientRequestCoroutine->setDowngrade(function ()
        {
            return 123;
        });
    });
    $this->http_output->end($result);
}
```

微服务调用中都自带熔断器，比如我们进行一个1~10000000加法运算，并设置超时时间为1s，我们访问这个服务会变得很慢，如果超过了超时时间将发生熔断。熔断会使得服务访问快速失败，熔断持续1s后会开放多次访问如果有一次超时则依旧熔断。

通过setDowngrade可以设置降级函数，如果发生熔断则触发降级函数，返回降级函数的值，如果发生超时也会访问降级函数。

通过熔断手段可以达到有效保护后台服务器的目的。

我们通过浏览器访问http://localhost:8081/TestController/testConsul3，会返回49999995000000，这时我们打开ab压测

```
ab -k -n100000 -c100 http://localhost:8081/TestController/testConsul3
```

在压测的过程中我们继续通过浏览器访问，可以发现返回的是123.

熔断器分为3个状态：

第一个状态是正常状态，这时所有的访问都能通过，如果在这过程中出现失败的情况，根据检查时间计算出一个比例这个比例达到一个阈值是会进入断路状态，

第二个状态是断路状态，这时所有的访问都不能通过经过断路尝试时间后会进入尝试状态，

第三个状态是尝试状态，这时熔断器会开放几次访问来确定服务是否恢复，如果这几次访问都成功通过那么将变成正常状态，如果有一次未通过则进入熔断状态。

默认熔断器设置的参数为：

阈值THRESHOLD=0.01；

检查时间CHECKTIME=2000ms

尝试打开的间隔TRYTIME=1000ms

尝试次数TRYMAX=3

可以在consul配置中修改断路器参数：

```
/**熔断器设置***/  
//阈值  
$config['fuse']['threshold'] = 0.01;  
//检查时间  
$config['fuse']['checktime'] = 2000;  
//尝试打开的间隔  
$config['fuse']['trytime'] = 1000;  
//尝试多少个  
$config['fuse']['trymax'] = 3;
```

集群-Cluster

集群-Cluster

SD框架支持集群模式部署，配置很简单。

- 安装以及配置consul-server和consul-client-agent，具体方法参考[微服务-Consul](#)
- 在consul.php配置文件中配置好相关信息，参考前面的配置说明。
- 在consul.php配置文件中打开Cluster的使能，并配置好端口。
- 启动多个SD服务器时需要保证Cluster的端口保持一致。

```
# consul.php

//是否开启TCP集群,启动consul才有用
$config['cluster']['enable'] = true;
//TCP集群端口
$config['cluster']['port'] = 9999;
```

无需多余的ip配置，一切的服务发现均依赖于Consul,部署好集群后，可以通过

```
http://localhost:8081/Status
```

查看服务器节点状态，可以查看微服务的发布状态。

```
{
  "start_time": 1520237652,
  "connection_num": 5,
  "accept_count": 17,
  "close_count": 12,
  "tasking_num": 0,
  "request_count": 5,
  "worker_request_count": 2,
  "now_task": [],
  "consul_services": {
    "TestController": [
      {
        "ID": "Http_TestController",
        "Service": "TestController",
        "Tags": [
          "http"
        ],
        "Address": "192.168.8.57",
        "Port": 8081,
        "EnableTagOverride": false,
        "CreateIndex": 1003694,
```

```

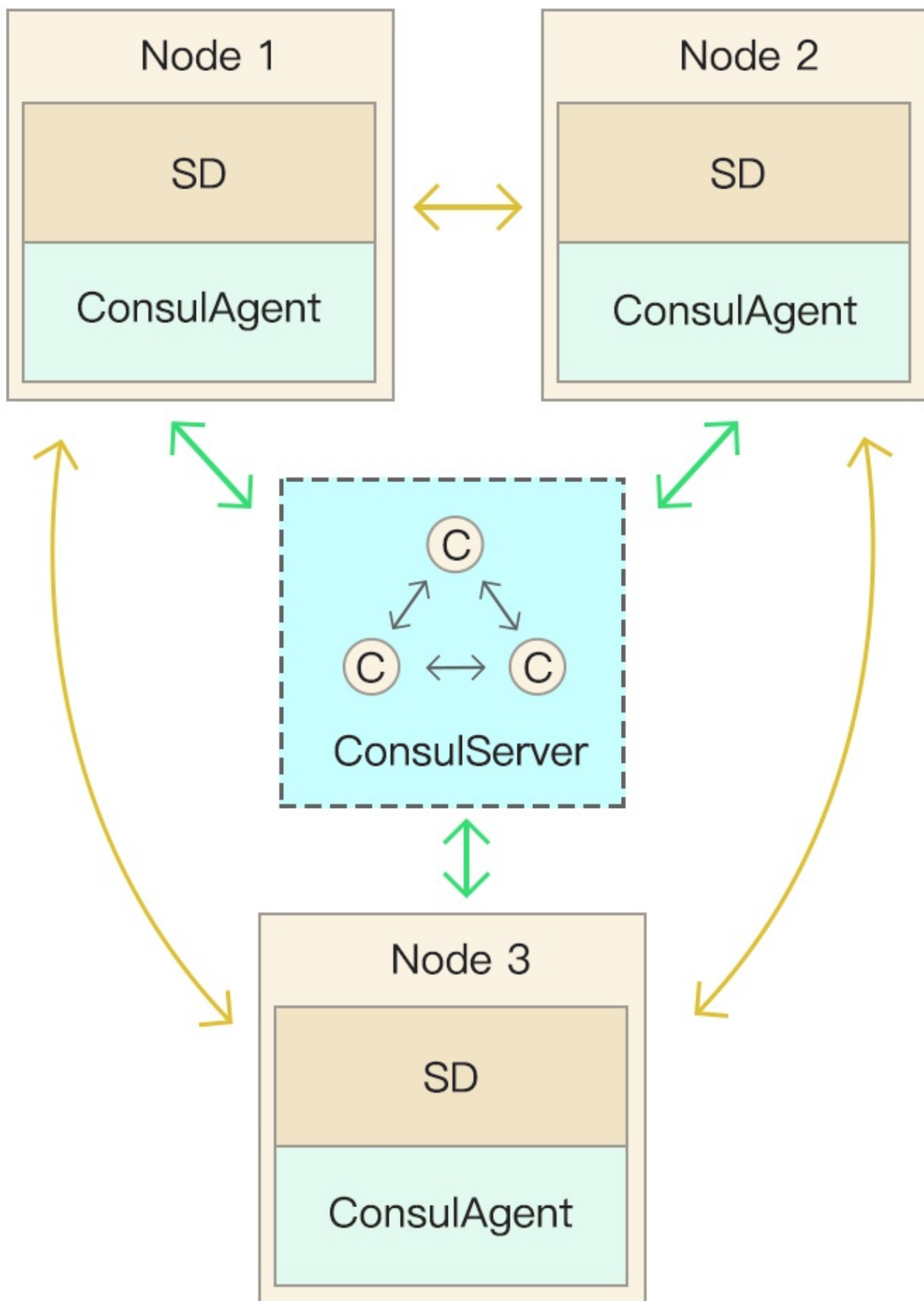
        "ModifyIndex": 1003699
    },
    {
        "ID": "Tcp_TestController",
        "Service": "TestController",
        "Tags": [
            "tcp"
        ],
        "Address": "192.168.8.57",
        "Port": 9091,
        "EnableTagOverride": false,
        "CreateIndex": 1003693,
        "ModifyIndex": 1003700
    }
],
"MathService": [
    {
        "ID": "Http_MathService",
        "Service": "MathService",
        "Tags": [
            "http"
        ],
        "Address": "192.168.8.57",
        "Port": 8081,
        "EnableTagOverride": false,
        "CreateIndex": 1003692,
        "ModifyIndex": 1003695
    },
    {
        "ID": "Tcp_MathService",
        "Service": "MathService",
        "Tags": [
            "tcp"
        ],
        "Address": "192.168.8.57",
        "Port": 9091,
        "EnableTagOverride": false,
        "CreateIndex": 1003691,
        "ModifyIndex": 1003701
    }
]
},
"cluster_nodes": [
    "SD-1"
],
"uidOnlineCount": 0
}

```

SD框架中除了CatCache外均已支持集群，使用时和单机一致。

关于集群

TCP/Websocket才需要集群部署，Http服务可以通过LVS或者Nginx实现。



高速缓存-CatCache

高速缓存-CatCache

- [高速缓存-CatCache](#)
 - [配置](#)
 - [使用](#)
 - [注意](#)

SD自带的可落地的高速缓存

配置

在CatCache.php配置文件中启用CatCache

```
/**
 * 是否启动CatCache
 */
$config['catCache']['enable'] = true;
//自动存盘时间
$config['catCache']['auto_save_time'] = 1000;
//落地文件夹
$config['catCache']['save_dir'] = BIN_DIR . '/cache/';
//RPC代理
$config['catCache']['rpcProxyClass'] = CatCacheRpcProxy::class;
//分割符
$config['catCache']['delimiter'] = ".";
auto_save_time 自动存盘时间，单位是秒
save_dir 存盘路径
rpcProxyClass RPC代理，高级用法
delimiter 路径分隔符
CatCache拥有2种存盘模式，一种是全盘落地通过auto_save_time参数控制，一旦到达时间就会执行存盘操作。
另外是日志模式，记录用户的增加删除操作，2种模式会尽量保证数据不会丢失。
```

delimiter是分隔符

CatCacheRpcProxy::getRpc()['test.a']

这样其实是获取test数组中a的值，通过更改delimiter可以变化分隔符，不影响数据。

auto_save_time要好好设计，不能太久，会导致log日志过多，太短影响效率。

使用

```
public function http_testSC1()
```

```

{
    // $result = isset(CatCacheRpcProxy::getRpc()['test.bc']); 协程不支持这种写法
    $result = CatCacheRpcProxy::getRpc()->offsetExists('test.bc');
    $this->http_output->end($result, false);
}

public function http_testSC2()
{
    unset(CatCacheRpcProxy::getRpc()['test.a']);
    $this->http_output->end(1, false);
}

public function http_testSC3()
{
    CatCacheRpcProxy::getRpc()['test.a'] = ['a' => 'a', 'b' => [1, 2, 3]];
    $this->http_output->end(1, false);
}

public function http_testSC4()
{
    // $result = CatCacheRpcProxy::getRpc()['test']; 协程不支持这样
    $result = CatCacheRpcProxy::getRpc()->offsetGet('test');
    $this->http_output->end($result, false);
}

public function http_testSC5()
{
    $result = CatCacheRpcProxy::getRpc()->getAll();
    $this->http_output->end($result, false);
}

```

注意

协程不支持

```

$result = isset(CatCacheRpcProxy::getRpc()['test.bc']) ;
$result = CatCacheRpcProxy::getRpc()['test'];

```

支持unset，因为unset不需要返回值

所以上面的用法需要改为

```

$result = CatCacheRpcProxy::getRpc()->offsetExists('test.bc');
$result = CatCacheRpcProxy::getRpc()->offsetGet('test');

```


万物-Actor

万物-Actor

什么是Actor你可以把它当成一个运算单元，用于模拟任何的事与物，Actor存在于内存中分布在不同的进程，或者分布在不同的机器上，Actor间可以互相通讯，Controller，Model甚至Actor自身都可以创建一个Actor，Actor同样也可以调用Model和Task。

拿棋牌举例子，你可以为房间创建一个RoomActor，为房间里的每一个人创建一个PlayerActor，RoomActor负责处理一共有多少局，轮到谁出牌了，PlayerActor则可以负责玩家的一系列行为，甚至玩家掉线充当玩家的AI。

利用好Actor可以简化你游戏的逻辑，使其更加拟人，逻辑表达更加的清晰，使用Actor的哲学就是万物的交流靠通讯。

Actor支持集群

Actor原型

Actor原型

SD 仿照Actor模型原理实现了独特的Actor。

Actor模型

Actor模型为并行而生，简单说是为解决高并发的一种编程思路。在Actor模型中，主角是Actor，类似一种worker，Actor彼此之间直接发送消息，不需要经过什么中介，消息是异步发送和处理的。在Actor模式中，“一切皆是Actor”，所有逻辑或者模块均别看做Actor，通过不同Actor之间的消息传递实现模块之间的通信和交互。Actor模型描述了一组为了避免并发编程的常见问题的公理：

- 1.所有Actor状态是Actor本地的，外部无法访问。
- 2.Actor必须只有通过消息传递进行通信。
- 3.一个Actor可以响应消息:推出新Actor,改变其内部状态,或将消息发送到一个或多个其他参与者。
- 4.Actor可能会堵塞自己,但Actor不应该堵塞它运行的线程。

Actor的基础就是消息传递

使用Actor模型的好处：

- 事件模型驱动--Actor之间的通信是异步的，即使Actor在发送消息后也无需阻塞或者等待就能够处理其他事情
- 强隔离性--Actor中的方法不能由外部直接调用，所有的一切都通过消息传递进行的，从而避免了Actor之间的数据共享，想要观察到另一个Actor的状态变化只能通过消息传递进行询问
- 位置透明--无论Actor地址是在本地还是在远程机上对于代码来说都是一样的
- 轻量性--Actor是非常轻量的计算单元，只需少量内存就能达到高并发

Actors

一个Actor指的是一个最基本的计算单元。它能接收一个消息并且基于其执行计算。

这个理念很像面向对象语言，一个对象接收一条消息（方法调用），然后根据接收的消息做事（调用了哪个方法）。

Actors一大重要特征在于actors之间相互隔离，它们并不互相共享内存。这点区别于上述的对象。也就是说，一个actor能维持一个私有的状态，并且这个状态不可能被另一个actor所改变。

聚沙成塔

One ant is no ant, one actor is no actor.

光有一个actor是不够的，多个actors才能组成系统。在actor模型里每个actor都有地址，所以它们才能够相互发送消息。

Actor模型的思想是：把你的应用程序看作是由许多轻量的被称之为“Actor”的实体组成的，每个Actor只负责一个很小的任务，职责单一且清晰，复杂的业务逻辑会通过多个Actor之间相互协作来完成，比如委派任务给其他的Actor或者传递消息给协作者。

Actors做什么

当一个actor接收到消息后，它能做如下三件事中的一件：

- Create more actors; 创建其他actors
 - Send messages to other actors; 向其他actors发送消息
 - Designates what to do with the next message. 指定下一条消息到来的行为
- 前两件事比较直观，第三件却很有意思。

我之前说过一个actor能维持一个私有状态。「指定下一条消息来到做什么」意味着可以定义下条消息来到时的状态。更清楚地说，就是actors如何修改状态。

设想有一个actor像计算器，它的初始状态是数字0。当这个actor接收到add(1)消息时，它并不改变它原本的状态，而是指定当它接收到下一个消息时，状态会变为1。

Actor的创建

Actor的创建

- Actor的创建
 - 生命周期
 - initialization初始化
 - destroy销毁
 - 创建Actor
 - 是否存在Actor
 - 自动恢复
 - 保留信息
 - 状态机
 - Actor的定时器
 - 销毁Actor

首先Actor依赖CatCache，如果没有开启CatCache将无法使用Actor。

生命周期

Actor创建后，会常驻内存，在整个服务器生命周期中不会消亡，除非手动销毁Actor。

无论是服务器重启或者是Reload，Actor都会自动恢复，但是注意由于重启和Reload会结束进程，Actor里的临时变量都会被清除，除非使用特殊手段保存下来。

initialization初始化

创建好Actor成功后会调用Actor的initialization方法。

可以在initialization中创建事件的侦听等等。

```
public function initialization($name, $saveContext = null)
{
    //接收自己的消息
    EventDispatcher::getInstance()->add($this->messageId, function ($event) {
        $this->handle($event->data);
    });
    //接收管理器统一派发的消息
    EventDispatcher::getInstance()->add(self::SAVE_NAME . Actor::ALL_COMMAND, function ($event) {
        $this->handle($event->data);
    });
}
```

比如默认的initialization就创建了2个事件的侦听。

destroy销毁

在destroy中处理应该销毁的东西。

```
public function destroy()
{
    ProcessManager::getInstance()->getRpcCall(ClusterProcess::class)->my_removeActor
($this->name);
    EventDispatcher::getInstance()->removeAll($this->messageId);
    EventDispatcher::getInstance()->remove(self::SAVE_NAME . Actor::ALL_COMMAND, [$t
his, '_handle']);
    foreach ($this->timerIdArr as $id) {
        @\swoole_timer_clear($id);
    }
    $this->saveContext->destroy();
    Pool::getInstance()->push($this->saveContext);
    $this->saveContext = null;
    $this->timerIdArr = [];
    $this->beginId = 0;
    $this->nowAffairId = 0;
    parent::destroy();
}
```

比如默认销毁了所有的定时器，和事件的侦听。

创建Actor

```
Actor::create(TestActor::class, "Test1");
```

通过上面的命令创建一个Actor，TestActor将继承Actor。Test1为这个Actor的名称，可以为同样的Actor类创建不同的实例，名称在整个集群环境中都必须唯一，否则会报错。

可以利用try来包裹创建命令，如果报错则代表Actor已存在。

是否存在Actor

```
Actor::has("Test1");
```

自动恢复

通过Actor::create创建Actor后，Actor将持续存在，即使重启服务器或者Reload，Actor都会被自动拉

起。

```
> [RLUA] 已加载sadd_from_count脚本
> [SYS] 已开启代码热重载
> [CatCache] 已完成加载缓存文件
> [Actor] 自动恢复了2 个Actor。
```

可以通过下面的命令清除Actor

```
php start_swoole_server.php clear
[OK] Clear actor and timer callback success
```

保留信息

上面说了重启后的Actor将失去所有的信息，除非通过特殊手法保留下来。

通过saveContext可以保存信息，即使重启服务器这些信息也不会丢失。

```
$this->saveContext['test'] = ["a"=>"aaa"];
```

注意：

可以通过以下方法获取值

```
$data = $this->saveContext['test'] ['a']; //OK
```

但不能直接使用下面的方法进行赋值

```
$this->saveContext['test'] ['a']='b' //会报错
```

如果需要修改深层的值可以通过下面的方法

```
$this->saveContext->getData()["test"]["a"] = "b";
$this->saveContext->save();
```

如果用到getData()方法修改就必须调用save()方法，不然不会保存。

状态机

Actor用状态机保存状态，以便恢复执行。

```
$this->setStatus("type", "1");
```

通过setStatus方法保存一个状态，重写registStatusHandle，用作触发状态改变的处理逻辑。

```
public function registStatusHandle($key, $value)
{
    switch ($key) {
        case 'type':
            switch ($value) {
                case 1:
                    $this->tick(100, function () {
                        echo "1\n";
                    });
                    break;
            }
            break;
    }
}
```

这样如果服务器Reload或者重启状态不会丢失，状态对应的处理函数会在Actor拉起时自动执行。

Actor的定时器

- public function tick(\$ms, \$callback, \$user_param = null); //定时器
- public function after(\$ms, \$callback, \$user_param = null); //延时执行
- public function clearTimer(\$id); //清除定时器

Actor销毁的时候会自动销毁全部的定时器。

销毁Actor

- Actor::destroyActor(\$name); //销毁某一个Actor
- Actor::destroyAllActor(); //销毁所有Actor

Actor间的通讯

Actor间的通讯

- [Actor间的通讯](#)
 - [与Actor通讯](#)
 - [事务](#)
 - [异常](#)

与Actor通讯

```
$rpc = Actor::getRpc("Test2");  
$result = $rpc->test1();
```

与Actor通讯都是RPC形式，和调用一个函数一样。

事务

Actor与Actor通讯的时候可以开启一个事务，开启事务后Actor可以保证运行的顺序。

```
$rpc = Actor::getRpc("Test2");  
try {  
    $beginid = $rpc->beginCo();  
    $result = $rpc->test1();  
    $result = $rpc->test2();  
    //var_dump($result);  
    $result = $rpc->test3();  
    //var_dump($result);  
} finally {  
    //var_dump("finally end");  
    $rpc->end();  
}
```

- beginCo 开始一个事务
- end 结束一个事务

并发情况下可以保证test1，test2，test3是按顺序执行的，中间不会插入其他执行语句。

Actor事务并没有回滚的操作，如果需要回滚需要通过业务代码自行实现。

end必须要执行，否则actor会一直被堵塞，事务运行期间可以通过try捕获异常，通过finally保证一定会执行end。

还有一种使用事务的方法

```
$rpc = Actor::getRpc("Test2");
$beginid = $rpc->beginCo(function ()use($rpc)
{
    $result = $rpc->test1();
    $result = $rpc->test2();
    //var_dump($result);
    $result = $rpc->test3();
});
$this->http_output->end(1);
```

这种方法更为推荐，始终都会自动执行end，不用担心end没有写。

beginCo第一个回调是运行回调，第二个回调是出错回调。

异常

Actor-RPC中需要返回个异常直接通过Throw方法即可，调用方会获得这个异常。

消息派发-EventDispatcher

消息派发-EventDispatcher

- [消息派发-EventDispatcher](#)
 - [addOnceCoroutine](#)
 - [dispatch](#)
 - [dispathToWorkerId](#)
 - [randomDispatch](#)
 - [add](#)
 - [remove](#)
 - [removeAll](#)
 - [应用场景](#)
 - [集群](#)

消息分发器

```
/**
 * 事件处理
 */
public function http_getEvent()
{
    $data = EventDispatcher::getInstance()->addOnceCoroutine('unlock', function
(EventCoroutine $e) {
        $e->setTimeout(10000);
    });
    //这里会等待事件到达, 或者超时
    $this->http_output->end($data);
}

public function http_sendEvent()
{
    EventDispatcher::getInstance()->dispatch('unlock', 'hello block');
    $this->http_output->end('ok');
}
```

上述代码getEvent描述的是等待事件的到达，事件超时时间为10s。

sendEvent代表是一个事件的派发。

访问getEvent浏览器会一直等待10s直到超时，期间如果访问sendEvent那么getEvent接口会立即返回'hello block'。

addOnceCoroutine

```
function addOnceCoroutine($eventType)
```

等待一次事件，事件到达自动清除事件的监听。

dispatch

发送一次事件，支持集群模式。发送的data会自动序列化。

```
function dispatch($type, $data = null, $onlyMyWorker = false, $fromDispatch = false
)
```

- onlyMyWorker为true代表只在本进程派发事件。
- fromDispatch请保持一直为false。

dispathToWorkerId

派发给指定进程

```
function dispathToWorkerId($workerId, $type, $data = null)
```

randomDispatch

```
function randomDispatch($type, $data = null)
```

随机选取一个进程派发，只支持本地，不支持集群。

add

```
function add($type, $listener)
```

回调的方式监听一个事件，这种方式不会自动清除事件的关联，会永久的监听这个事件，直到手动移除。

remove

```
function remove($type, $listener)
```

移除一个事件的监听

removeAll

```
function removeAll($type = null)
```

移除这个类型的所有事件

应用场景

事件的应用场景非常灵活，可以实现高性能业务毫秒定时器，可以实现高性能堵塞方式的http请求，框架中很多通讯和RPC均应用到Event。

集群

Event的派发支持集群

例子A

例子A

延迟队列-TimerCallBack

延迟队列-TimerCallBack

游戏一般对消息队列有一定的使用需求，但一般的开源消息队列并没有很好的处理延时消息的派发。比如rabbitmq虽然它是可以通过死信队列来模拟延时队列但是不仅难用还有很大的限制。

SD提供了一个简单可靠的延迟消息队列模块TimerCallBack。

```
$token = TimerCallBack::addTimer(2,TestModel::class,'testTimerCall',[123]);
```

简单的通过下面一段代码就可以实现延迟2s执行TestModel中的testTimerCall函数。而且没有任何副作用，即使重启服务器延时队列也会很好的完成使命。

延迟队列需要开启CatCache

协程

协程

- [协程](#)
 - [创建协程环境](#)
 - [协程CoroutineBase](#)
 - [协程的并发](#)

SD3.0提供了内置的协程，不再需要yield，但为了更好的理解框架，我们还需要掌握些知识点。

创建协程环境

swoole提供了创建协程环境的方法，该方法用户在使用SD的过程中不需要使用。

```
go(function()  
{  
    //这里便可以执行协程，已经创建了协程的运行环境了。  
});
```

协程CoroutineBase

SD项目中用到协程的地方都使用了CoroutineBase基类，这个类是协程的基础。

通过CoroutineBase可以设置协程的各种参数，比如超时啊，无异常啊等等。

- setDelayRecv 设置延时Recv，这是需要手动调用recv才能获取结果
- dump dump信息
- setTimeout 设置超时时间
- setDowngrade 设置降级函数，超时也会触发降级函数返回函数的值
- noException 不返回超时异常，直接返回\$return的值

那么我们怎么获取这个CoroutineBase呢，基本所有的协程方法中都有个\$set参数设置一个回调函数，在回调函数中可以进行协程的设置。

```
$data = EventDispatcher::getInstance()->addOnceCoroutine('unlock', function (EventCo  
routine $e) {  
    $e->setTimeout(10000);  
});
```

比如Event将返回EventCoroutine，Mysql将返回MySQLCoroutine，Redis将返回RedisCoroutine，当然你也可以直接使用CoroutineBase。

协程的并发

如果出于效率考虑需要多个协程并发，那么需要做下面的操作。

```
$redis1 = $this->redis_pool->coroutineSend("get",['test'],function (RedisCoroutine $
redisCoroutine)
{
    $redisCoroutine->setDelayRecv();
});
$redis2 = $this->redis_pool->coroutineSend("get",['test'],function (RedisCoroutine $
redisCoroutine)
{
    $redisCoroutine->setDelayRecv();
});
$result1 = $redis1->recv();
$result2 = $redis2->recv();
```

设置了setDelayRecv后coroutineSend返回的将不是结果而是个协程对象，后面需要手动调用recv来获取到值，通过这么设置redis1和redis2请求是并发的。

订阅与发布

订阅与发布

- [订阅与发布](#)
 - [主题名和主题过滤器 Topic Names and Topic Filters](#)
 - [主题通配符 Topic wildcards](#)
 - [主题层级分隔符 Topic level separator](#)
 - [多层通配符 Multi-level wildcard](#)
 - [单层通配符](#)
 - [主题语义和用法 Topic semantic and usage](#)
 - [SD提供的API](#)

SD框架提供了一个集群的订阅发布功能，实现和MQTT订阅发布一致。

- `addSub($topic)`
- `removeSub($topic)`
- `sendPub($topic, $data, $destroy = true)`

首先订阅信息是保留在服务器内存中，重启服务器后订阅的主题会全部清空。

如果用户想实现群组功能，订阅与发布只能作为通讯的基础服务，流程如下。

- A用户加入了a，b，c三个群组，服务器需要负责将A的群组关系落地。
- 通过addSub为A用户订阅a，b，c三个群组对应的主题名
- A用户上线需要重新为其订阅a，b，c三个群组对应的主题名
- 订阅与发布服务并不知道群里有多少人它只负责订阅与发布，其余需要自己实现
- 可以通过主题通配符实现特殊订阅

主题名和主题过滤器 Topic Names and Topic Filters

主题通配符 Topic wildcards

主题层级（topic level）分隔符用于将结构化引入主题名。如果存在分隔符，它将主题名分割为多个主题层级 topic level。

订阅的主题过滤器可以包含特殊的通配符，允许你一次订阅多个主题。

主题过滤器中可以使用通配符，但是主题名不能使用通配符。

主题层级分隔符 Topic level separator

斜杠（`'/'` U+002F）用于分割主题的每个层级，为主题名提供一个分层结构。当客户端订阅指定的主

主题过滤器包含两种通配符时，主题层级分隔符就很有用了。主题层级分隔符可以出现在主题过滤器或主题名字的任何位置。相邻的主题层次分隔符表示一个零长度的主题层级。

多层通配符 Multi-level wildcard

数字标志（‘#’ U+0023）是用于匹配主题中任意层级的通配符。多层通配符表示它的父级和任意数量的子层级。多层通配符必须位于它自己的层级或者跟在主题层级分隔符后面。不管哪种情况，它都必须是主题过滤器的最后一个字符。

非规范评注

例如，如果客户端订阅主题 “sport/tennis/player1/#”，它会收到使用下列主题名发布的消息：

“sport/tennis/player1”

“sport/tennis/player1/ranking”

“sport/tennis/player1/score/wimbledon”

非规范评注

- “sport/#” 也匹配单独的 “sport”，因为 # 包括它的父级。
- “#” 是有效的，会收到所有的应用消息。
- “sport/tennis/#” 也是有效的。
- “sport/tennis#” 是无效的。
- “sport/tennis/#/ranking” 是无效的。

单层通配符

加号（‘+’ U+002B）是只能用于单个主题层级匹配的通配符。

在主题过滤器的任意层级都可以使用单层通配符，包括第一个和最后一个层级。然而它必须占据过滤器的整个层级。可以在主题过滤器中的多个层级中使用它，也可以和多层通配符一起使用。

非规范评注

例如，“sport/tennis/+” 匹配 “sport/tennis/player1” 和 “sport/tennis/player2”，但是不匹配 “sport/tennis/player1/ranking”。

同时，由于单层通配符只能匹配一个层级，“sport/+” 不匹配 “sport” 但是却匹配 “sport/”。

非规范评注

- “+” 是有效的。
- “+/tennis/#” 是有效的。
- “sport+” 是无效的。
- “sport/+/player1” 也是有效的。
- “/finance” 匹配 “+/+” 和 “/+”，但是不匹配 “+”。

主题语义和用法 Topic semantic and usage

主题名和主题过滤器必须符合下列规则：

- 所有的主题名和主题过滤器必须至少包含一个字符。
- 主题名和主题过滤器是区分大小写的。
- 主题名和主题过滤器可以包含空格。
- 主题名或主题过滤器以前置或后置斜杠 “/” 区分。
- 只包含斜杠 “/” 的主题名或主题过滤器是合法的。
- 主题名和主题过滤器不能包含空字符 (Unicode U+0000) 。
- 主题名和主题过滤器是UTF-8编码字符串，它们不能超过65535字节。
-

除了不能超过UTF-编码字符串的长度限制之外，主题名或主题过滤器的层级数量没有其它限制。

匹配订阅时，服务端不能对主题名或主题过滤器执行任何规范化（normalization）处理，不能修改或替换任何未识别的字符。

主题过滤器中的每个非通配符层级需要逐字符匹配主题名中对应的层级才算匹配成功。

非规范评注

使用UTF-8编码规则意味着，主题过滤器和主题名的比较可以通过比较编码后的UTF-8字节或解码后的Unicode字符。

非规范评注

- “ACCOUNTS” 和 “Accounts” 是不同的主题名。
- “Accounts payable” 是合法的主题名
- “/finance” 和 “finance” 是不同的。

如果订阅的主题过滤器与消息的主题名匹配，应用消息会被发送给每一个匹配的客户端订阅。主题可能是管理员在服务端预先定义好的，也可能是服务端收到第一个订阅或使用那个主题名的应用消息时动态添加的。服务端也可以使用一个安全组件有选择地授权客户端使用某个主题资源。

SD提供的API

- `getSubMembersCountCoroutine` 获取Topic的数量
- `getSubMembersCoroutine` 获取Topic的Member
- `getUidTopicsCoroutine` 获取uid的所有订阅

以上API使用`get_instance()`获取

- `addSub($topic)`

- `removeSub($topic)`
- `sendPub($topic, $data,$excludeUids = [], $destroy = true)` `$excludeUids` 排除的uid , 在这个数组的uid将接收不到订阅消息

以上API在Controller中

MQTT简易服务器

MQTT简易服务器

SD提供一个简易的MQTT服务器。

使用范围

- 支持MQTT协议(3/3.0/3.1/3.1.1)
- 支持完善的订阅和发布规则
- 支持集群
- 支持客户端验证
- 仅仅支持QOS1模式
- 仅仅支持TCP协议
- 不支持保留消息

如何使用

打开Ports配置添加一个端口，使用MqttPack的封装器

```
$config['ports'][] = [  
    'socket_type' => PortManager::SOCK_TCP,  
    'socket_name' => '0.0.0.0',  
    'socket_port' => 9092,  
    'pack_tool' => 'MqttPack',  
    'route_tool' => 'NormalRoute',  
    'middlewares' => ['MonitorMiddleware']  
];
```

MqttPack将会把Mqtt的消息路由至MqttController控制器，如果想实现自定义，那么请参照MqttController书写规范，并重写MqttPack指定到你所设置的控制器中。

MqttController

以下方法对应MQTT的控制命令

- connect
- publish
- pubrel
- subscribe
- unsubscribe

- pingreq
- disconnect

这些是必须实现的方法，其中在connect中实现客户端的用户密码的验证。

AMQP异步任务调度

AMQP异步任务系统

需要了解的知识。

- [进程管理](#)
- [RabbitMQ](#)

本系统由SD框架和RabbitMQ搭建。

创建异步作业进程

通过继承AMQPTaskProcess，我们来创建一个异步任务作业的进程类。

```
class MyAMQPTaskProcess extends AMQPTaskProcess
{

    public function start($process)
    {
        parent::start($process);
        //获取一个channel
        $channel = $this->connection->channel();
        //创建一个队列
        $channel->queue_declare("msgs");
        //框架默认提供的路由，也可以自己写
        $this->createDirectConsume($channel, 'msgs');
        //等待所有的channel
        $this->connection->waitAllChannel();
    }

    /**
     * 路由消息返回class名称
     * @param $body
     * @return string
     */
    protected function route($body)
    {
        return TestAMQPTask::class;
    }

    protected function onShutDown()
    {
        // TODO: Implement onShutDown() method.
    }
}
```

通过createDirectConsume函数可以快速创建一个消费队列。

- createDirectConsume

```
function createDirectConsume($queue, $prefetch_count = 2, $global = false, $exchange
    = null, $consumerTag = null)
```

一般情况我们只需要设置queue和prefetch_count这两个参数。

queue为消费队列的名称，prefetch_count=2代表这个队列只能被这个进程同时消费2次，直到消费成功或者失败，简单的来说并发为2。

global参数代表这个并发是针对队列还是进程的。false是针对队列，true代表是进程。

我们可以多次调用createDirectConsume来消费不同的队列。

- route

route路由的作用，\$body是消费得到的值，这个函数需要返回一个class名。

创建作业任务

创建类继承AMQPTask。

```
class TestAMQPTask extends AMQPTask
{
    /**
     * @var TestModel
     */
    public $TestModel;

    public function initialization(AMQPMessage $message)
    {
        parent::initialization($message);
        $this->TestModel = $this->loader->model(TestModel::class, $this);
    }

    /**
     * handle
     * @param $body
     */
    public function handle($body)
    {
        var_dump($body);
        $this->ack();
    }
}
```

- initialization

和Model，Controller一样用于初始化，或者进行loader

- handle
处理任务，处理任务一定需要调用ack或者是reject。
- ack
任务处理完毕
- reject

```
function reject($requeue = true)
```

任务被拒绝，requeue=true代表这个任务回到队列，false代表任务被抛弃。

创建用户进程

在AppServer中创建进程

```
/**
 * 用户进程
 */
public function startProcess()
{
    parent::startProcess();
    for ($i=0;$i<5;$i++)
    {
        ProcessManager::getInstance()->addProcess(MyAMQPTaskProcess::class,$i);
    }
}
```

这样我们创建了5个异步任务进程。

注意

1. 消费队列必须存在，否则会报错
2. 一定在handle处理结束后调用ack或者reject

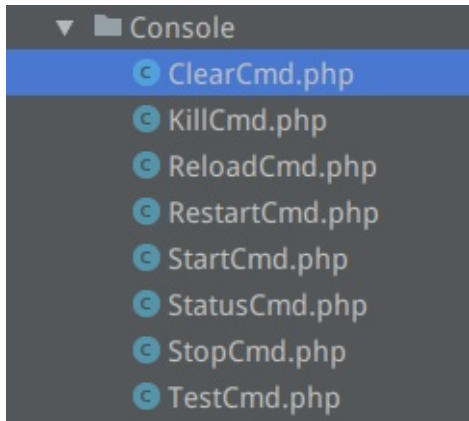
自定义命令-Console

自定义命令-Console

SD集成了Symfony\Component\Console，可以标准化的创建命令。

命令目录

框架的命令目录在Server/Console下，开发者编写自己的命令时可以参考。



自定义目录在app/Console下，放在这个文件夹下的命令将会被自动加载。

添加命令

Step1：需要继承Symfony\Component\Console\Command\Command类。

Step2：在configure函数中设置命令的名称和描述

```
protected function configure()
{
    $this->setName('clear')->setDescription("Clear server actor and timer callback");
}
```

Step3：实现execute函数，设置命令的动作

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $io = new SymfonyStyle($input, $output);
    new AppServer();
    $process = new CatCacheProcess("", "");
    $process->clearActor();
    $process->clearTimerBack();
    $process->autoSave();
    $io->success("Clear actor and timer callback success");
}
```

完成以上步骤就可以运行自己的命令了。

查看命令

进入项目的bin目录下，可以执行list查看命令

```
php start_swoole_server.php list
Console Tool

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet            Do not output any message
  -V, --version          Display this application version
  --ansi                Force ANSI output
  --no-ansi              Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 f
or more verbose output and 3 for debug

Available commands:
  clear    Clear server actor and timer callback
  help     Displays help for a command
  kill     Kill server
  list     Lists commands
  reload   Reload server
  restart  Restart server
  start    Start server
  status   Server Status
  stop     Stop(Kill) server
  test     Test case
```

你编写的命令也将出现在里面。

调试工具Channel

调试工具-Channel

3.2.0版本提供一个新的调试工具channel，该工具可以将服务器与客户端的交互完美复制到控制器中打印出来。

准备工作

1.server.php配置中开启下面配置

```
//是否允许监控流量数据
$config['allow_MonitorFlowData'] = true;
```

2.backstage.php配置中

```
//是否启用backstage
$config['backstage']['enable'] = true;
```

如果没有backstage二进制程序的开发者开启这个选项会报错。这里提供另一个开启方式，当然也可以加入VIP获得监控系统。

在ports.php配置中手动开启一个18083端口（如果有backstage二进制文件直接配置backstage.php不需要这一步）

```
$config['ports'][] = [
    'socket_type' => PortManager::SOCK_WS,
    'socket_name' => '0.0.0.0',
    'socket_port' => 18083,
    'route_tool' => 'ConsoleRoute',
    'pack_tool' => 'ConsolePack',
    'event_controller_name' => Server\Components\Backstage\Console::class,
    'connect_method_name' => "onConnect",
    'close_method_name' => "onClose",
    'method_prefix' => 'back_',
    'opcode' => PortManager::WEBSOCKET_OPCODE_TEXT,
    'middlewares' => ['MonitorMiddleware', 'NormalHttpMiddleware']
];
```

用法

```
php start_swoole_server.php channel -u 1
```

一定要有-u参数后面要接需要跟踪的uid名称。

可以添加需要过滤的多个参数

```
php start_swoole_server.php channel -u 1 cmd:123 cmd:124
```

上面将打印出协议中包含cmd=>123或cmd=>124的消息

```
test@ubuntu:/home/catGame/bin$ php start_swoole_server.php channel -u 1 cmd:204
$SYS_CHANNEL/1/recv
Array
(
    [req] =>
    [token] => 0
    [cmd] => 204
    [code] => 0
    [msg] =>
)
$SYS_CHANNEL/1/send
Array
(
    [cmd] => 204
    [token] => 0
    [code] => 1
    [rep] => 1120
)
^C
```

特别注意事项

特别注意事项

这里总结一些SD框架使用过程中的特别注意的地方。

不要通过git clone方式部署代码

不要通过git clone方式部署代码，除非你对框架源码很了解，通过composer安装框架。

特别注意不要使用sleep，die，exit这类方法

框架是多进程模式，以上的命令会导致严重的进程奔溃问题，sleep会使得进程堵塞。

不要使用new创建Model

不要new一个Model，应该使用loader->model方法载入一个model。同样也不能直接new一个Task。

不要在__construct里使用loader

不要在__construct里使用loader，loader应该在initialization里使用。

Actor内的异常需要自己捕获

Actor默认没有进行异常捕获，一旦抛出的异常均是未捕获的异常，会导致进程退出重启。

日常问题总结

总结日常问题

为什么启动TCP服务器后客户端发送消息会断开连接

客户端发送的消息必须符合服务器定义的协议规范,tcp协议和http协议不同,tcp协议需要自定义协议规范,告诉服务器如何分割消息,常见的分割消息的方法有eof和length.如果客户端随意发送消息,将会被服务器踢下线.

inotify报错upper limit on inotify watches reached

在对一个大磁盘进行inotify监听时,爆出如下错误:

```
Failed to watch /mnt/;  
upper limit on inotify watches reached!  
Please increase the amount of inotify watches allowed per user via
```

cat一下/proc/sys/fs/inotify/max_user_watches,默认值是8192,执行下面的语句即可

```
echo 8192000 > /proc/sys/fs/inotify/max_user_watches
```

swSocket_set_buffer_size

```
> [RLUA] 已加载sadd_from_count脚本.  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(8, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(9, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(10, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(11, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(12, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] ERROR swSocket_set_buffer_size(:412): setsockopt(13, SOL_SOCKET, SO_SNDBUF, 134217728) failed. Error: No buffer space available[55].  
[2018-03-06 14:19:06 @81932.0] WARNING swServer_start_check: serv->max_connection is exceed the maximum value[4864].
```

一般出现在虚拟机或者mac系统中。

如果出现上面的错误,请手动进入server.php配置中,将max_connection字段修改到1024或者更小。

线上版本需要把max_connection设置到足够的大,一般100000,代表支持100000个连接。

[CoroutineTask]: Time Out!,[Request]: [Event][PR]

```
[CoroutineTask]: Time Out!, [Request]: [Event][PR]3->9:1
```

出现Event[PR]这种错误,如果你是使用虚拟机那么很抱歉,虚拟机中Swoole的进程间通讯功能有可能不能使用,如果想用完整的SD功能那么请使用docker或者物理机。

出现Class not found

如果出现这个错误，请检查命名空间问题，一般情况是命名空间不对，或者不符合psr/4规范

实践案例

实践案例

通过一些案例，介绍SD强大之处。

物联网自定义协议

物联网自定义协议

SD在物联网上有成熟产品，物联网协议一般是二进制协议，这里提供一个案例帮助大家理解自定义协议。

我们先观看一个数据结构文档

一：数据格式

头部分	数据部分	校验部分
11字节	Key+值…（ N字节 ）	2字节

二：头部分

起始	长度	版本号	设备号	指令
1个字节	1个字节	1个字节	7个字节	1个字节

字段	占用字节	说明
起始	1	起始字节：平台向终端下发，起始字节为：0xFE 终端向平台上发，起始字节为：0xEF
长度	1	数据部分字节总数
版本号	1	协议版本号，当前版本v1.0（ 0x01 ）
设备号	7	10字节设备号
指令	1	数据帧指令，具体见下表

OK，可以了通过上面的数据结构文档我们已经可以进行解包操作了。

解包

第一步：找寻合适的解包方法

我们需要把上面的协议和SD的俩种解包方法对比一下，看看到底用哪种方法解包。
EOF方法不适合上面的协议，因为不是按照固定字段分割各个协议的。
Len方法适合上面的协议，因为可以通过长度分割各个协议包，协议中也提供了长度字段。
所以我们确定使用LenPack作为封装器，我们可以找到框架的LenJsonPack把它作为参考来设计我们自己的Pack。

第二步：重新组合成符合SD审美观点的数据结构

上面的文档提供的数据结构我们人类很好理解，但是对于SD来说并不符合它的审美标准。

SD的审美标准是协议头仅仅只需要包含长度部分，其他全部都属于协议体

那么我们根据SD的审美重新组合下协议，如下：

协议头部分：

起始	长度
1个字节	1个字节

协议体部分：

版本号	设备号	指令	数据部分	校验部分
1个字节	7个字节	1个字节	(N字节)	2字节

接下来我们的目标就是帮助SD找到这个长度字段，首先这个长度字段代表的仅仅是数据部分的长度，这点是需要我们清楚知道的，否则将会导致最后解析错误。

这里先介绍4个重要的字段意思

- \$package_length_type
长度值的类型，接受一个字符参数，与php的 pack 函数一致。目前Swoole支持10种类型：
c：有符号、1字节
C：无符号、1字节
s：有符号、主机字节序、2字节
S：无符号、主机字节序、2字节
n：无符号、网络字节序、2字节
N：无符号、网络字节序、4字节
l：有符号、主机字节序、4字节（小写L）
L：无符号、主机字节序、4字节（大写L）
v：无符号、小端字节序、2字节
V：无符号、小端字节序、4字节
- \$package_length_type_length 上面字段的长度，比如C就是1，N是4

- \$package_length_offset
length长度值在包头的第几个字节
- \$package_body_offset
从第几个字节开始计算长度，一般有2种情况：
length的值包含了整个包（包头+包体），package_body_offset 为0
包头长度为N字节，length的值不包含包头，仅包含包体，package_body_offset设置为N

看完上面的介绍我们开始确定这些字段的值：

package_length_type：上面协议长度为1那么我们这里可以设置为c

package_length_type_length：对应为1

package_length_offset：这里为1，因为长度字段前面只有1个字节

package_body_offset：这里应该填写13，上面协议的长度仅仅包含数据部分所以我们要加上前面的11字节和末尾的2个字节。

第三步：开始写Pack

我们拷贝一份LenJosnPack，重命名后我们先修改上面4个字段的参数。

Pack总共有4个重要的解包相关的函数。

- encode 加上协议头
- decode 去除协议头
- pack 封装协议
- unPack 解析协议

现在我们根据上面符合SD审美的协议结构进行书写这4种方法。

客户端发来的消息经过Swoole分离成各个独立消息后首先进入unPack中，也就是说unPack传递进去的是完整的协议，然后通过decode去除协议头，传递给路由器进一步处理。

服务器下发的消息首先进入pack函数，这个消息是可能是php的一个类型，pack把这个类型转换成协议体然后调用encode为其添加协议头，最后传递给客户端。

```
/**
 * 数据包编码加上协议头
 * @param $buffer
 * @return string
 * @throws SwooleException
 */
public function encode($buffer)
{
    //|版本|设备|指令|数据|
    //| 0 |1 ~ 7|8|9~9+n|
    //获取长度
    $length = strlen($buffer) - 9;
    //制作校验码
    $check = '';
```

```

    //|起始|长度|版本|设备|指令|数据|校验
    //| 0 | 1 | 2 |3 ~ 9|10|11~11+n|11+n+1~11+n+3|
    return hex2bin('f0') . pack($this->package_length_type, $length).$buffer . $check;
}

/**
 * 去除协议头
 * @param $buffer
 * @return string
 */
public function decode($buffer)
{
    //|起始|长度|版本|设备|指令|数据|校验
    //| 0 | 1 | 2 |3 ~ 9|10|11~11+n|11+n+1~11+n+3|
    //去掉前2个字节（起始和长度）
    $ret = substr($buffer, $this->package_length_type_length + 1);
    //获取后俩个校验字节
    $crc = substr($ret, 0, -2);
    //这里进行校验失败抛异常
    //去除校验字节
    $ret = substr($ret, 0, strlen($ret) - 2);
    return $ret;
}
// 封装协议体
public function pack($data, $topic = null)
{
    return $this->encode(hex2bin($data));
}
// 解析协议体
public function unPack($data)
{
    $data = $this->decode($data);
    //|版本|设备|指令|数据|
    //| 0 |1 ~ 7|8|9~9+n|
    $version = $data[0];
    $device_sn = substr($data,1,7);
    $command = $data[8];
    $receivedData = substr($data, 9);
    $data = [];
    $data['device_sn'] = $device_sn;
    $data['command'] = $command;
    $data['data'] = $receivedData;
    return $data;
}

```

大家注意二进制协议解析要养成这种注释的习惯，会让你的思路更加清晰，不会犯错。

Actor在游戏的应用

Actor在游戏的应用

以棋牌游戏举个例子 希望大家能通过简单的例子来了解SD强大的功能！

第一：创建游戏模型

首先需要知道几点：

- 1、Actor的name是全局唯一不可重复
- 2、在Actor中，每个Actor对应的\$this->saveContext（可储存信息[详情点击](#)）都是私有独立的。
- 3、Actor必须只有通过消息传递进行通信。

更多关于actor介绍请查看[Actor原型](#)

以下为实例代码：

需要在app/Actors下创建2个actor模型

1、RoomActor.php 房间actor

```
<?php
/**
 * 游戏房间
 * User: 4213509@qq.com
 * Date: 18-3-12
 * Time: 上午10:24
 */
namespace app\Actors;
use app\GameException;
use Server\Components\Event\EventDispatcher;
use Server\CoreBase\Actor;
use Server\CoreBase\ChildProxy;

class RoomActor extends Actor
{
    /**
     * 初始化储存房间信息
     * @param $room_info
     */
    public function initData($room_info){

        $this->saveContext['info'] = $room_info;

    }

    /**
     * 进房询问
```

```

    * @param $user_info
    */
    public function joinRoomReply($user_info){
        //代码下文有详细介绍
    }

}
?>

```

2、PlayerActor.php 用户actor

```

<?php
/**
 * 游戏房间
 * User: 4213509@qq.com
 * Date: 18-3-12
 * Time: 上午10:24
 */
namespace app\Actors;
use app\GameException;
use Server\Components\Event\EventDispatcher;
use Server\CoreBase\Actor;
use Server\CoreBase\ChildProxy;

class PlayerActor extends Actor
{

    public function initialization($name, $saveContext = null)
    {
        yield parent::initialization($name, $saveContext); // TODO: Change the autogenerated stub

        // 已经初始化过的要订阅离线事件消息 每次重启都会执行, 确保不丢失
        if (isset($this->saveContext['info']['userId'])) {
            yield $this->offlineHandle($this->saveContext['info']['userId']);
        }
    }

    /**
     * 初始化储存用户信息
     * @param $user_info
     */
    public function initData($user_info){

        $this->saveContext['info'] = $room_info;

    }
}

```

```
}
?>
```

第二：创建房间

```
$RoomActorName = 'roomActorId' . $room_id; // 房间的actor可以用一个规则来命名来保证唯一性

Actor::create(RoomActor::class, $RoomActorName); //创建房间Actor
Actor::getRpc($RoomActorName)->initData($room_info); // 可以在创建完成后在初始化房间的数据
```

这样一个房间actor的创建和初始化就已经完成了，也可以将房间的actorName存入房间信息表中，这样再次访问可以判断是否存在房间ActorName，存在就不创建直接使用，不存在房间ActorName的话再创建。

第三：用户进房

房间创建完毕后，用户开始进房间，首先需要询问房间是否可进！

实例代码如下：

```
$join_res =Actor::getRpc($RoomActorName)->joinRoomReply($user_info);
```

在房间的actor模型中，joinRoomReply负责检查该房间是否已经满了&&用户是否已经进来了。检查后没有问题，就告诉用户可以进房间。

于是、用户开始在房间里占了个位进来了，然后创建一个自己的actor和初始化一些自己的数据储存在\$this->saveContent中

在actor中，每个\$this->saveContent都是独立私有的，用来储存当前角色的属性

实例代码如下：

```
public function joinRoomReply($user_info)
{
    $user_id = $user_info['id'];
    $join_users = $this->saveContext['user_list']; // 已经进入的用户储存在这里
    if (!isset(join_users[$user_id])) { // 检查用户是否已经进来
        //当前用户还没有进入房间的逻辑
        if (count($join_users) >= 6) { //房间总共可以进入6个人，查询是否已经满了。满了
            就不让进来了
            throw new GameException("房间已满");
        }
        $corrent_user_actor = $this->name . $user_id; //根据规则创建一个唯一的命名
```



```

        try {
            Actor::create(PlayerActor::class, $corrent_user_actor); //没有进来过
创建一个用户的actor
            $user_info['room_actor'] = $this->name;
            Actor::getRpc($corrent_user_actor)->initData($user_info); //初始化该a
ctor的默认值属性 initData这个方法是储存用户信息
        } catch (\Exception $e) {
            throw new GameException('创建失败：' . $e->getMessage())
        }
        $this->saveContext->getData()['user_list'][$user_id] = $corrent_user_act
or; //储存当前用户和其中对应的actorName
        $this->saveContext->save();
    }else{
        //重新进入房间的逻辑
    }
    get_instance()->addSub('Room/' . $this->name, $user_id); // 进入成功，开始订阅
当前房间消息
    get_instance()->pub('Room/' . $this->name, '给所有用户推送$user_id进到房间了');
    return true;
}

```

第四：进行游戏

待续...

第五：用户掉线逻辑

1、AppController.php 默认控制器，连接、断开时间处理控制器

```

<?php

namespace app\Controllers;

use Server\Components\Event\EventDispatcher;
use Server\CoreBase\Controller;

/**
 * Created by PhpStorm.
 * User: 4213509@qq.com
 * Date: 18-05-17
 * Time: 上午10:10
 */
class AppController extends Controller
{

```

```

public function onClose()
{
    if ($this->uid > 0) {
        $this->unBindUid();
        print_r($this->uid."用户断线");
        EventDispatcher::getInstance()->dispatch('offline' . $this->uid);
    }
    $this->destroy();
}

..... 省略

?>

```

2、PlayerActor.php 用户actor

```

<?php
/**
 * 游戏房间
 * User: 4213509@qq.com
 * Date: 18-05-17
 * Time: 上午10:10
 */
namespace app\Actors;
use app\GameException;
use Server\Components\Event\EventDispatcher;
use Server\CoreBase\Actor;
use Server\CoreBase\ChildProxy;

class PlayerActor extends Actor
{

    public function initialization($name, $saveContext = null)
    {
        yield parent::initialization($name, $saveContext); // TODO: Change the autogenerated stub

        // 已经初始化过用户信息的要订阅离线事件消息 每次重启都会执行这里，确保不丢失
        if (isset($this->saveContext['info']['userId'])) {
            yield $this->offlineHandle($this->saveContext['info']['userId']);
        }
    }

    // 事件处理函数
    public function offlineHandle($userId)
    {
        // 监听事件消息
        EventDispatcher::getInstance()->add('offline' . $userId, yield function ($event) use ($userId) {
            $user_Info = $this->saveContext['info'];

```

```

        $room_id = $this->saveContext['info']['room_id'];
        //取消订阅房间消息
        get_instance()->removeSub("Room/"$room_id ., $userId);
        //删除当前监听的事件
        EventDispatcher::getInstance()->removeAll('offline' . $userId);
        //通知房间该用户掉线 或者是退出 例如 $code =1 退出 $code =2 头像暗了
        Actor::getRpc($user_Info['room_actor']->offlineMessage($user_Info['user
Id'], $code));
    });
}

/**
 * 初始化储存用户信息
 * @param $user_info
 */
public function initData($user_info){

    $this->saveContext['info'] = $room_info;

}

}
?>

```

3、RoomActor.php 房间actor

```

<?php
/**
 * 游戏房间
 * User: 4213509@qq.com
 * Date: 18-05-17
 * Time: 上午10:10
 */
namespace app\Actors;
use app\GameException;
use Server\Components\Event\EventDispatcher;
use Server\CoreBase\Actor;
use Server\CoreBase\ChildProxy;

class RoomActor extends Actor
{

    // 用户离线消息处理 自己判断是要让用户退出还是离线
    public function offlineMessage($userId, $code = 1)
    {

        //处理离线逻辑
        //....
    }
}

```

```

        // 发送消息给房间所有人该用户的离线事件
        $send['cmd'] = $code;
        $send['code'] = 200;
        $send['data']['userId'] = $userId;
        $this->sendPub('Room/' . $this->saveContext['room_id'], $send);
    }

    /**
     * 初始化储存房间信息
     * @param $room_info
     */
    public function initData($room_info){

        $this->saveContext['info'] = $room_info;

    }

    /**
     * 进房询问
     * @param $user_info
     */
    public function joinRoomReply($user_info)
    {
        $user_id = $user_info['id'];
        $join_users = $this->saveContext['user_list']; // 已经进入的用户储存在这里
        if (!isset($join_users[$user_id])) { // 检查用户是否已经进来
            //当前用户还没有进入房间的逻辑
            if (count($join_users) >= 6) { //房间总共可以进入6个人, 查询是否已经满了。
                满了就不让进来了
                throw new GameException("房间已满");
            }
            $corrent_user_actor = $this->name . $user_id; //根据规则创建一个唯一的命
            名
            try {
                Actor::create(PlayerActor::class, $corrent_user_actor); //没有进来
                过 创建一个用户的actor
                $user_info['room_actor'] = $this->name;
                Actor::getRpc($corrent_user_actor)->initData($user_info); //初始
                化该actor的默认值属性 initData这个方法是储存用户信息
            } catch (\Exception $e) {
                throw new GameException('创建失败: '.$e->getMessage())
            }
            $this->saveContext->getData()['user_list'][$user_id] = $corrent_user
            _actor; //储存当前用户和其中对应的actorName
            $this->saveContext->save();
        }else{
            //重新进入房间的逻辑
        }
        get_instance()->addSub('Room/' . $this->name, $user_id); // 进入成功, 开始
        订阅当前房间消息
        get_instance()->pub('Room/' . $this->name, '给所有用户推送$user_id进到房间了
        ');
    }

```

```
        return true;
    }

}
?>
```

Mongodb以及一些同步扩展的使用

Mongodb以及一些同步扩展的使用

我们知道worker进程是异步的，如果在worker进程中出现了同步阻塞的方法就会导致整个服务器堵塞。Mongodb是个很常用的扩展，但可惜的是没有异步方法，如果在worker进程中大量使用Mongodb的扩展方法，会导致服务器的吞吐量有限。

那么我们怎么样更好的使用这类扩展呢？答案是用Task。

Task是swoole提供的同步进程，worker进程和Task进程是通过异步通讯的，我们通过这一特性可以将Mongodb放在Task进程中使用，这样一来我们只需要开启多个Task就可以模拟达到异步连接池的效果，当然性能肯定不及异步，但是服务器整体的吞吐量得到了保证。

onSwooleWorkerStart

在这里我们找到Task进程，并为Task进程创建Mongodb连接。

```
protected $mongodb;
public function onSwooleWorkerStart($serv, $workerId)
{
    parent::onSwooleWorkerStart($serv, $workerId);
    if($this->isTaskWorker()){
        $this->mongodb = new MongoClient();
        //连接
    }
}

public function getMongoDb()
{
    return $this->mongodb();
}
```

`$this->isTaskWorker()`判断当前是Task进程，我们worker进程是不要创建Mongodb连接的。

Task

```
class TestTask extends Task
{
    protected $mongoDb;
    public function __construct()
    {
        parent::__construct();
        $this->mongoDb = get_instance()->getMongoDb();
    }
}
```

```
    }  
}
```

然后使用这个Test就和使用Model一样的道理。

```
public function http_testTask()  
{  
    $testTask = $this->loader->task(TestTask::class, $this);  
    $result = $testTask->test();  
    $this->http_output->end($result);  
}
```

在server.php配置中我们把Task进程多开几个即可。

开发者工具

开发者工具

SD在未来版本中将提供各种各样的实用的开发者工具。

开发者工具将以命令行的方式集成在SD发布包中使用。

```
Console Tool

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi        Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  channel  channel monitor
  clear    Clear server actor and timer callback
  help     Displays help for a command
  init     Init PlayGround
  kill     Kill server
  list     Lists commands
  model    Test Model
  proto    Proto build
  reload   Reload server
  restart  Restart server
  start    Start server
  status   Server Status
  stop     Stop(Kill) server
  test     Test case
```

接下来我将分别介绍这些工具包的使用

Channel

channel命令需要指定一个uid作为源，命令启动后将会和服务器建立一个信息通道，该通道会自动复制uid客户端与服务器信息交互中产生的所有流量，并经过筛选显示到控制面板中。

```
php start_swoole_server.php channel -u 1
```

-u后面是绑定的uid，这是必不可少的参数。

通道将会监听该uid与服务器交互的流量，并以PHP数组形式显示到控制面板中。


```

$SYS_CHANNEL/1/recv
Array
(
    [req] => Array
        (
            [id] => 1
            [nickname] => 蛇蛇
            [uid] => 1
            [sex] => 2
            [ear] => 10001
            [eye] => 10023
            [face] => 10040
            [pattern] => 10060
            [leg] => 10079
            [birth] => 2018-05-16 09:40:25
            [luck] => 18
            [charm] => 21
            [f_id] => 0
            [m_id] => 0
            [ff_id] => 0
            [fm_id] => 0
            [mf_id] => 0
            [mm_id] => 0
            [gene_lim] => 8
            [procreation_lim] => 14400
            [event_cd_lim] => 4
            [clear_lim] => 12000
            [hungry_lim] => 12000
            [fatigue_lim] => 100
            [shit_speed] => 1
            [hungry_speed] => 2
            [dirty_speed] => 2
            [gold_speed] => 8
            [cat_level] => 1
            [cat_character] => Array
                (
                    [0] => 16
                    [1] => 21
                    [2] => 32
                    [3] => 37
                    [4] => 47
                    [5] => 43
                    [6] => 72
                )
            )
        )

```

`$SYS_CHANNEL/1/recv` 代表是服务器接收到客户端的信息，相反的`$SYS_CHANNEL/1/send` 代表是服务器发送给客户端的信息

可以看到这是基于SD消息订阅实现的功能。SD的订阅发布系统功能强大性能卓越，该命令可以安全的调试线上服务器，而不用担心性能。

`-u 1` 后面可以继续接上多个参数，这些参数用于过滤，只有符合参数条件的消息才会被打印到控制台上。

```
php start_swoole_server.php channel -u 1 cmd:401
```

每个参数的格式都是`$key:$value`型，在消息数组结构中只要符合这种对应关系就会被选中，多个参数是或的关系。

有了这个工具开发者可以指定查看客户端数据流信息进行错误定位，无需再苦苦和客户端联调，特别是线上环境。

Proto

Proto工具用于生成私有协议，如果你不满足protobuf或者其他协议类型，或者觉得直接使用json或者msgpack开发上不便捷，或者其他特殊原因。那么Proto工具是个最棒的选择。

目前Proto工具可以生成JS和PHP的协议，并且完美的与SD框架结合，让你无脑编写业务代码，爽到不要。

首先我们有个协议文档，这个文档是基于XML的。我们所有协议的定义都通过这个XML定义。

- Controller模块

```
</controller>
<controller name="CShop" cmd="6">
  <method name="GetShopList" cmd="1" req="EmptyData" rep="ShopInfo[]" des="获取商店列表"/>
  <method name="Buy" cmd="2" req="int" rep="EmptyData" des="购买"/>
  <method name="LevelUp" cmd="3" req="int" rep="FurnitureInfo" des="升级，req填家具type"/>
</controller>
<controller name="CMail" cmd="7">
```

和SD的Controller一摸一样，就是控制器，在这里定义控制器和方法，cmd是作为协议号。

- Struct结构体模块

```
<struct class="MailInfo" des="邮件信息">
  <bean name="id" type="string" des="id"/>
  <bean name="uid" type="int" des="uid"/>
  <bean name="create_time" type="int" des="创建时间"/>
  <bean name="msg" type="string" des="信息"/>
  <bean name="item1_id" type="int" des="id"/>
  <bean name="item1_count" type="int" des="数量"/>
  <bean name="item2_id" type="int" des="id"/>
  <bean name="item2_count" type="int" des="数量"/>
  <bean name="item3_id" type="int" des="id"/>
  <bean name="item3_count" type="int" des="数量"/>
  <bean name="item4_id" type="int" des="id"/>
  <bean name="item4_count" type="int" des="数量"/>
  <bean name="item5_id" type="int" des="id"/>
  <bean name="item5_count" type="int" des="数量"/>
  <bean name="is_read" type="bool" des="是否已读"/>
  <bean name="is_receive" type="bool" des="是否接收"/>
</struct>
```

定义所有的结构体。

结构体可以继续当type使用，甚至可以继承。

```
<struct class="AccountRep" des="登录信息">
  <bean name="userInfo" type="UserInfo" des="用户信息"/>
  <bean name="furnitureInfos" type="FurnitureInfo[]" des="家具信息"/>
  <bean name="catInfos" type="CatInfo[]" des="猫咪信息"/>
</struct>
```

- Marco模块

```
<marco name="CONNECT_HOST_MSG" type="string" value="192.168.86.90:9001" des="MSG连接地址"/>
<marco name="CONNECT_HOST_JSON" type="string" value="192.168.86.90:9002" des="JSON连接地址"/>
```

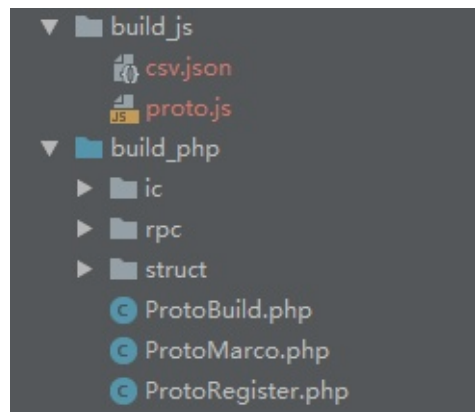
用于定义常量

整个协议都是基于上面3大模块构建的。

编写好XML后我们通过下面的命令就可以生成协议代码了，协议生成器拥有一个模板代码，开发者通过修

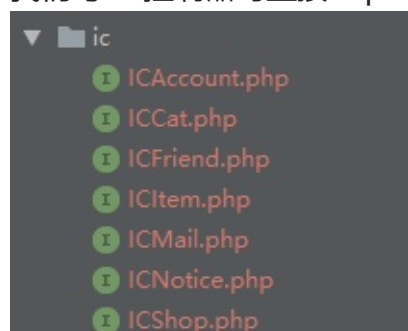
改模板代码可以生成自己自定义的协议，生成器也是通过PHP写的，更高级的需求可以直接修改生成器。

我们来看看生成的代码吧。



PHP包含3个文件夹ic/rpc/struct,JS就简单了只有一个proto.js。

我们写SD控制器时直接implements对应的接口就能自动生成对应的控制器方法了，相当的简单。



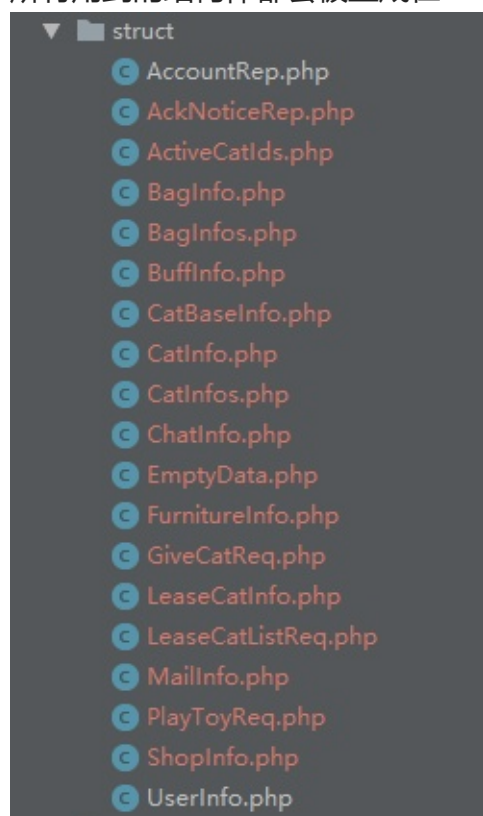
```
use ...

class CAccount extends BaseController implements ICAccount
{
    /**
     * @param CAccount_Login_101 $data
     * 用户信息请求
     * @return mixed
     */
    public function Login(CAccount_Login_101 $data)
    {
        // TODO: Implement Login() method.
    }

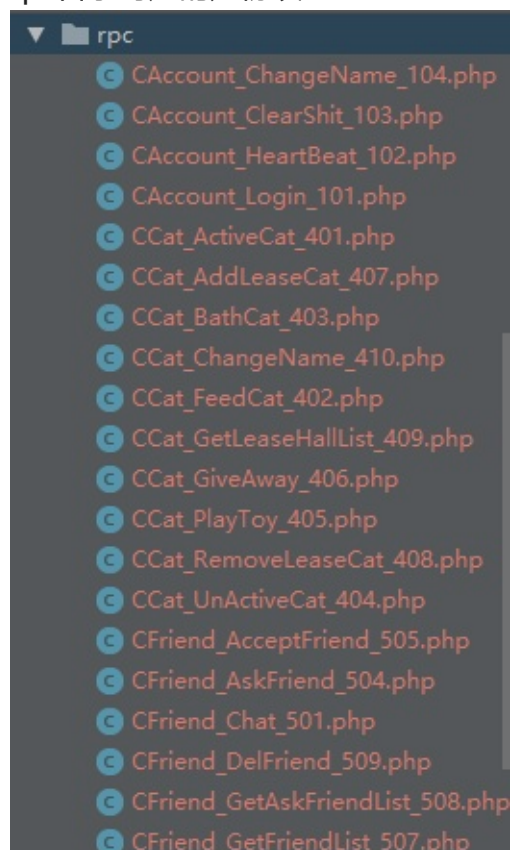
    /**
     * @param CAccount_HeartBeat_102 $data
     * 心跳
     * @return mixed
     */
    public function HeartBeat(CAccount_HeartBeat_102 $data)
    {
        // TODO: Implement HeartBeat() method.
    }

    /**
     * @param CAccount_ClearShit_103 $data
     * 清理shit
     * @return mixed
     */
    public function ClearShit(CAccount_ClearShit_103 $data)
    {
        // TODO: Implement ClearShit() method.
    }
}
```

所有用到的结构体都会被生成在struct目录中



rpc目录对应的是协议



Model模块

Model模块是用于调试Model的，它可以直接访问Model对象通过控制台调试。

```
php start_swoole_server.php model
```

下面就是过程引导：




```

[22] getCatCount
[23] changeMaster
[24] getCatCount
> 12

输入参数 catId 的值 (或者输入保存的变量名以 $ 开头):
> 1

输入参数 uid 的值 (或者输入保存的变量名以 $ 开头):
>

输出结果:
proto\struct\CatInfo Object
(
  [catBaseInfo] => proto\struct\CatBaseInfo Object
    (
      [id] => 1
      [nickname] => 小白
      [uid] => 1
      [sex] => 2
      [ear] => 10002
    )
)

```

首先会先让你输入model名称，然后会显示出所有的方法，选择方法后输入对应的变量。

protected,private,public的方法都会被显示出来，都可以被调用，但是protected,private方法中如果存在协程切换会有bug出现。

如果遇到参数是个复杂对象无法通过控制台输入怎么办？

我们可以通过调用对应Model将结果保存成临时变量使用。

```

[process_time] => 1525748329
[status] => 1
[status_pass_time] => 8600
[is_on_lease] =>
[buffInfos] => Array
(
)
)

将保存到变量，输入变量名以 $ 开头，或者 Ctrl-C 结束程序：
> $c1

```

输入参数 `catInfoA` 的值 (或者输入 保存的变量名以 `$` 开头):

> `$c1`

输入参数 `catInfoB` 的值 (或者输入 保存的变量名以 `$` 开头):

> `$c2`

! [NOTE] 注意: 此函数是非 `Public` 函数, 通过反射调用如果有协程切换不会返回结果

输出结果:

```
proto\struct\CatInfo Object
(
  [catBaseInfo] => proto\struct\CatBaseInfo Object
    (
      [id] =>
      [nickname] => 仔猫
      [uid] =>
      [sex] => 2
```

通过此工具开发过程中测试Model是不是变得很轻松了呢。