目錄

SwooleDistributed 使用手册

本手册适用于SwooleDistributed最新版本。

项目代码仓库：SwooleDistributed

qq交流群：569037921

前言:化繁为简，从零开始的框架设计

前言

经历了一个又一个项目，也接触了很多的PHP框架，我欣赏CI的简约，又贪婪swoole的效率，我将CI和swoole很草率的结合到了一起。起初呢风平浪静，慢慢的就遇到了不少的瓶颈，毕竟CI的设计理念还是贴合FPM模式，如何更加得心应手的使用swoole，同时追求开发上和运行时的效率呢，最主要的还是要方便扩展，就萌生了SwooleDistributed这个开源框架的想法。

在SwooleDistributed发布之前，开源社区还没有过针对swoole的分布式框架，起初的目的并不是一个完整的应用框架，而是一个简单的分布式通讯框架，后来需求变多了，框架也就慢慢的丰满了。

分布式

分布式这东西并不是有多神秘，但一个框架在基础构思中就包含分布式的思想，那无疑方便对以后的扩展。分布式系统涉及到多太物理机之间的调控，配置起来也是较为麻烦，SwooleDistributed使用了内网发现的技术手段，自动发现集群环境的物理机进行连接，简化了配置，甚至达到了无配置。

SwooleDistributed在前期可以控制成本的使用单机模式进行部署，也可以在后期进行水平扩展，对逻辑代码无需任何的改动。

你所需要的就是多增加物理机，跑上服务器就行啦。

MVCT

解决了底层的分布式通讯问题，接下来就是MVC结构的搭建了，这部分借鉴了CI的设计，使用Loader模块加载对应的Model，通过路由访问对应的Controller。相信使用过CI框架的工程师很容易就上手。
此外引入了Swoole独特的Task，将swoole的Task进行了封装优化，更加易于使用。

```
//TestTask.php
class TestTask extends Task
{
    public function test()
    {
        print_r("test timer task\n");
    }
}
//TestController.php
class TestController extends Controller
{
  public function http_test_task()
    {
        $task = $this->loader->task('TestTask');
        $task->test();
        $result = yield $task->coroutineSend();
        $this->http_output->end($result);
    }
}
```

通过定时器和Task的结合，开发者可以很方便的制作定时任务，而这一切只需要简单的配置即可。

```
/**
 * timerTask定时任务
 * （必填）task名称 task_name
 * （必填）执行task的方法 method_name
 * （选填）执行区间 [start_time,end_time) 格式： Y-m-d H:i:s 没有代表一直执行
 * （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
 */
$config['timerTask'][] = [
    'task_name'=>'TestTask',
    'method_name'=>'test',
    'start_time'=>'Y-m-d 00:00:00',
    'end_time'=>'Y-m-d 23:59:59',
    'interval_time'=>'2',
];
```

异步连接池

swoole提供了redis和mysql的异步客户端，大大提高的服务端的效率，但问题又来了，如果使用异步客户端，就必须维护一个异步连接池。
SwooleDistributed设计了一个通用的连接池模块，通过这个模块可以快捷简单的创建客户端的连接池，不仅如此redis和mysql的连接池早已封装在核心代码中，开发者只需调用其中的Api无需关注连接池的维护。
swoole的异步redis客户端是基于hredis和我们通常使用的redis扩展不一样，在使用批量方法时回调的值是有所区别，SwooleDistributed屏蔽了这一点不同之处，使用起来和平常的redis扩展一致。
异步的mysql客户端事务通常是比较难写的，SwooleDistributed同样针对mysql事务进行了封装，使用起来也非常的方便。
此外mysql也提供了一个语法构建器，可以简单方便的构建mysql语法。

```
 public function test_coroutine()
    {
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('uid', 10303)->coroutineSend();
        $result = yield $mySqlCoroutine;
        $redisCoroutine = $this->redis_pool->coroutineSend('get'
, 'test');
        $result = yield $redisCoroutine;
        return $result;
    }
```

协程

之前我去过很多家公司，交流的过程中发现有些公司使用php做短连接，用golang做长连接，我问他们为何不用swoole呢？回答都是异步回调太难写了。 确实异步回调写起来很不好看，可能会有多层回调的嵌套，复杂点的代码非常的难看，
swoole2.0已经使用了协程，但首先是新功能稳定性尚且不知，其次不支持php7，于是我呢就对现有框架进行了一次大的调整，通过yield关键字实现了全异步的协程风格。使用起来非常的简单，和一般的写法没有什么太大的区别，只要遵循一个原则，涉及到异步的地方加上yield关键字就可以。

```
    /**
     * 协程测试
     */
    public function http_testCoroutine()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $result = yield $this->testModel->test_coroutine();
        $this->http_output->end($result);
    }
```

协程之间支持嵌套，也支持throw Exception异常捕获，总之用起来和同步的写法基本一致。 协程之间不存在堵塞，也可以通过控制yield的位置调整send和rev的调度策略。 比如：

```
$mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select('*')
->from('account')->where('uid', 10303)->coroutineSend();
$result = yield $mySqlCoroutine;
$redisCoroutine = $this->redis_pool->coroutineSend('get', 'test'
);
$result = yield $redisCoroutine;
```

这段代码调度策略是mysql_send->mysql_rev->redis_send->redis_rev;
我们调整下yield的位置

```
$mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select('*')
->from('account')->where('uid', 10303)->coroutineSend();
$redisCoroutine = $this->redis_pool->coroutineSend('get', 'test'
);
$result = yield $mySqlCoroutine;
$result = yield $redisCoroutine;
```

这段代码调度策略变成了mysql_send->redis_send->mysql_rev->redis_rev;

同时支持TCP和HTTP

这样你就可以开启一个服务同时处理TCP和HTTP请求了，代码复用率就高很多了，同时SwooleDistributed还提供了一些策略，方便隔离http和tcp的路由请求。

Protobuf

提供了一个Protobuf的RPC实例，其实你可以通过框架的Pack和Route模块自由扩展，提供这个例子无非是我自己项目的需要顺便就发出来了，当然你可以自己实现。

用户案例

（DOER）做到

官网：http://www.zdoer.net/

简介： 做到Doer是一款针对高校学生群体、全国领先的S2C互助帮忙平台， 由深圳市有我行科技有限公司开发并由北京市做到团队推广和运营， 旨在为大学生提供便捷校园生活， 做学生专属的校园小秘书。

点聊

官网：http://dianliaoapp.com/

简介：点聊是一个免费匿名语音聊天交友软件，任何人都能在这里迅速匹配到倾诉的对象，排解寂寞、治愈心灵。点聊，超便捷的聊天工具！超刺激的交友平台！敢聊、爱聊的小伙伴快来，这里有一堆热心陪聊的小鲜肉等着你来电~打电话、唱歌、恋爱、假装情侣、弹幕吐槽...想怎么玩就怎么玩！点聊，点聊app，app，一点

聊，点聊语音，电话，语音，软件，电话软件，聊天，交友，社交，匿名，偷窥，偷听，点聊官网，语音聊天，主播，寂寞，空虚，夜聊，交友软件，免费交友，赚钱，速配

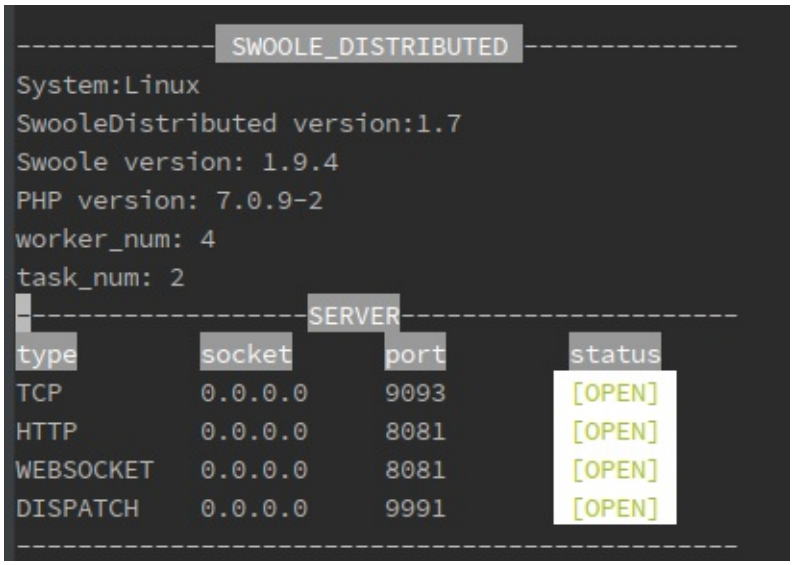
附录：框架性能报告

服务器基准

系统：Deepin 15.3 Ubuntun内核

CPU：Inter I3 4核

内存：8G



Http请求ab压测数据

ab和服务器在同一物理机上

```
/**
     * http测试
     */
    public function http_test()
    {
        $this->http_output->end('helloworld', false);
    }
```

```
ab -c100 -n1000000 -k http://localhost:8081/TestController/test
```

```
Server Software:        swoole-http-server
Server Hostname:        localhost
Server Port:            8081

Document Path:          /TestController/test
Document Length:        10 bytes

Concurrency Level:      100
Time taken for tests:   9.035 seconds
Complete requests:      1000000
Failed requests:        0
Keep-Alive requests:    1000000
Total transferred:      163000000 bytes
HTML transferred:       10000000 bytes
Requests per second:    110676.31 [#/sec] (mean)
Time per request:       0.904 [ms] (mean)
Time per request:       0.009 [ms] (mean, across all concurrent requests)
Transfer rate:          17617.42 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       1
Processing:     0    1   1.6      0      29
Waiting:        0    1   1.6      0      29
Total:          0    1   1.6      0      29

Percentage of the requests served within a certain time (ms)
  50%      0
  66%      1
  75%      1
  80%      1
  90%      1
  95%      3
  98%      7
  99%      9
 100%     29 (longest request)
```

Http请求Redis压测数据

单一请求

```
/**
     * http redis 测试
     */
    public function http_redis()
    {
        $value = $this->redis_pool->getCoroutine()->get('test');
        $this->http_output->end(1, false);
    }
    /**
     * http 同步redis 测试
     */
    public function http_aredis()
    {
        $value = get_instance()->getRedis()->get('test');
        $this->http_output->end(1, false);
    }
```

测试协程方法

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

```
Server Software:        swoole-http-server
Server Hostname:        localhost
Server Port:            8081

Document Path:          /TestController/redis
Document Length:        1 bytes

Concurrency Level:      100
Time taken for tests:   2.466 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    100000
Total transferred:      15300000 bytes
HTML transferred:       100000 bytes
Requests per second:    40546.17 [#/sec] (mean)
Time per request:       2.466 [ms] (mean)
Time per request:       0.025 [ms] (mean, across all concurrent requests)
Transfer rate:          6058.17 [Kbytes/sec] received
```

测试同步方法

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

```
Server Software:        swoole-http-server
Server Hostname:        localhost
Server Port:            8081

Document Path:          /TestController/aredis
Document Length:        1 bytes

Concurrency Level:      100
Time taken for tests:   2.718 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    100000
Total transferred:      15300000 bytes
HTML transferred:       100000 bytes
Requests per second:    36786.83 [#/sec] (mean)
Time per request:       2.718 [ms] (mean)
Time per request:       0.027 [ms] (mean, across all concurrent requests)
Transfer rate:          5496.47 [Kbytes/sec] received
```

协程方式比同步方式更加快速

多数请求

```
/**
     * http redis 测试
     */
    public function http_redis()
    {
        $value = $this->redis_pool->getCoroutine()->get('test');
        $value1 = $this->redis_pool->getCoroutine()->get('test1')
;
        $value2 = $this->redis_pool->getCoroutine()->get('test2')
;
        $value3 = $this->redis_pool->getCoroutine()->get('test3')
;
        yield $value;
        yield $value1;
        yield $value2;
        yield $value3;
        $this->http_output->end(1, false);
    }
    /**
     * http 同步redis 测试
     */
    public function http_aredis()
    {
        $value = get_instance()->getRedis()->get('test');
        $value1 = get_instance()->getRedis()->get('test1');
        $value2 = get_instance()->getRedis()->get('test2');
        $value3 = get_instance()->getRedis()->get('test3');
        $this->http_output->end(1, false);
    }
```

测试协程方法

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

```
Server Software:        swoole-http-server
Server Hostname:        localhost
Server Port:            8081

Document Path:          /TestController/redis
Document Length:        1 bytes

Concurrency Level:      100
Time taken for tests:   7.375 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    100000
Total transferred:      15300000 bytes
HTML transferred:       100000 bytes
Requests per second:    13560.00 [#/sec] (mean)
Time per request:       7.375 [ms] (mean)
Time per request:       0.074 [ms] (mean, across all concurrent requests)
Transfer rate:          2026.05 [Kbytes/sec] received
```

测试同步方法

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

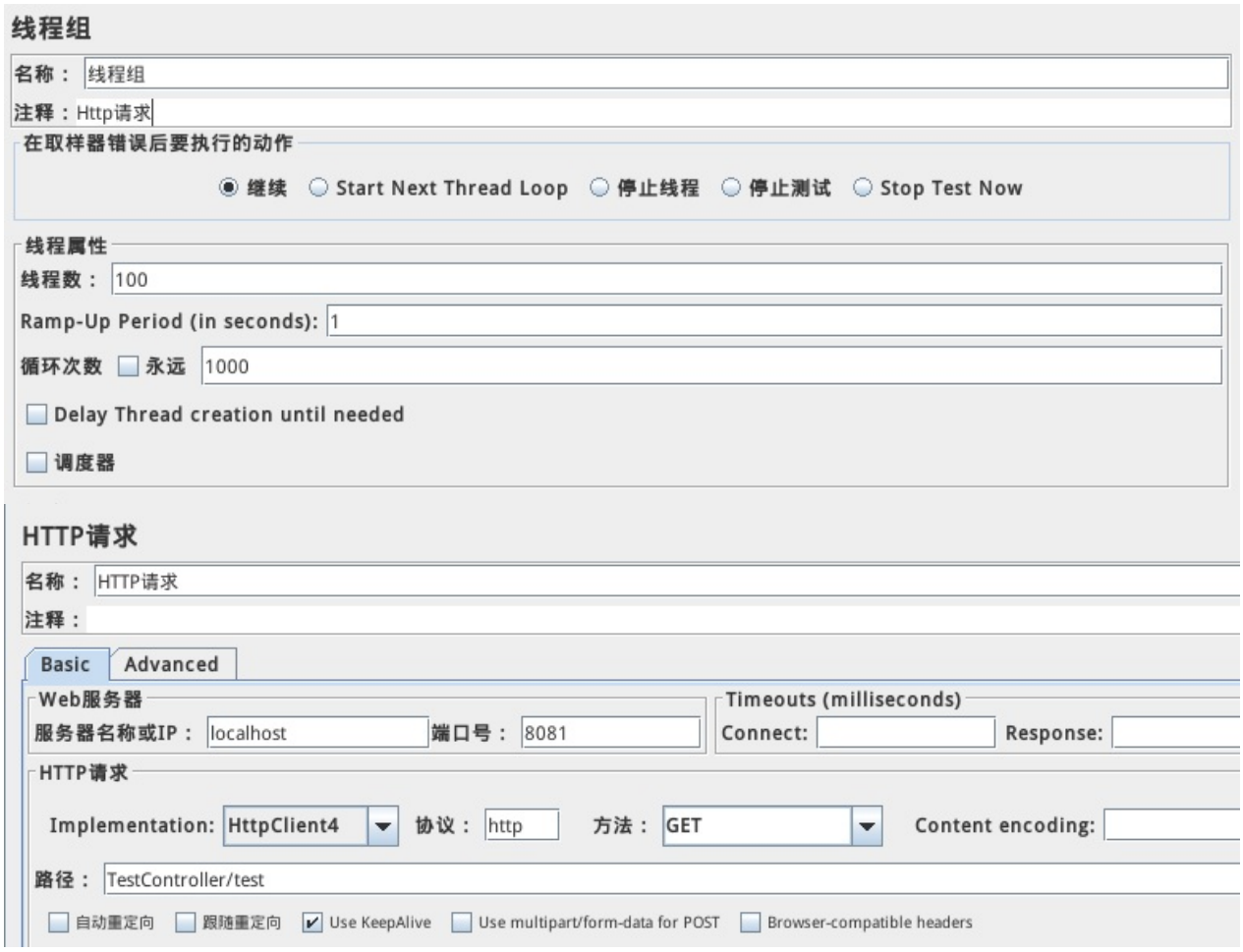
```
Server Software:        swoole-http-server
Server Hostname:        localhost
Server Port:            8081

Document Path:          /TestController/aredis
Document Length:        1 bytes

Concurrency Level:      100
Time taken for tests:   8.312 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    100000
Total transferred:      15300000 bytes
HTML transferred:       100000 bytes
Requests per second:    12030.99 [#/sec] (mean)
Time per request:       8.312 [ms] (mean)
Time per request:       0.083 [ms] (mean, across all concurrent requests)
Transfer rate:          1797.60 [Kbytes/sec] received
```

同样协程的方法更加快速。

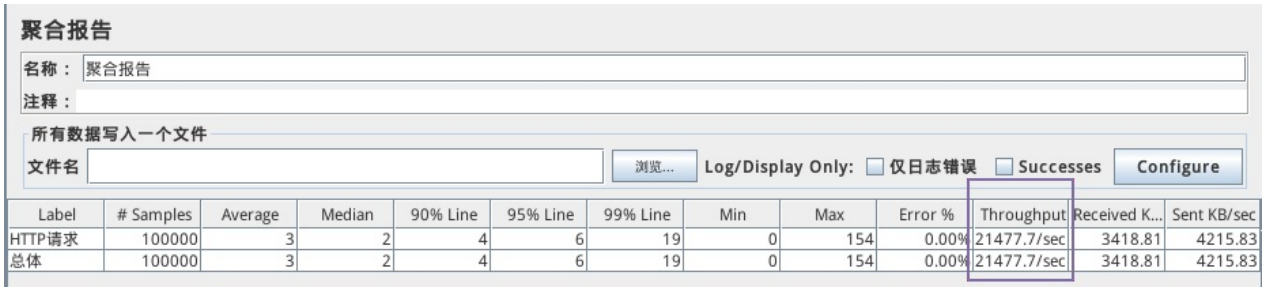
Http请求的JMeter聚合报告

测试访问的路径为TestController/test

配置：



聚合报告：



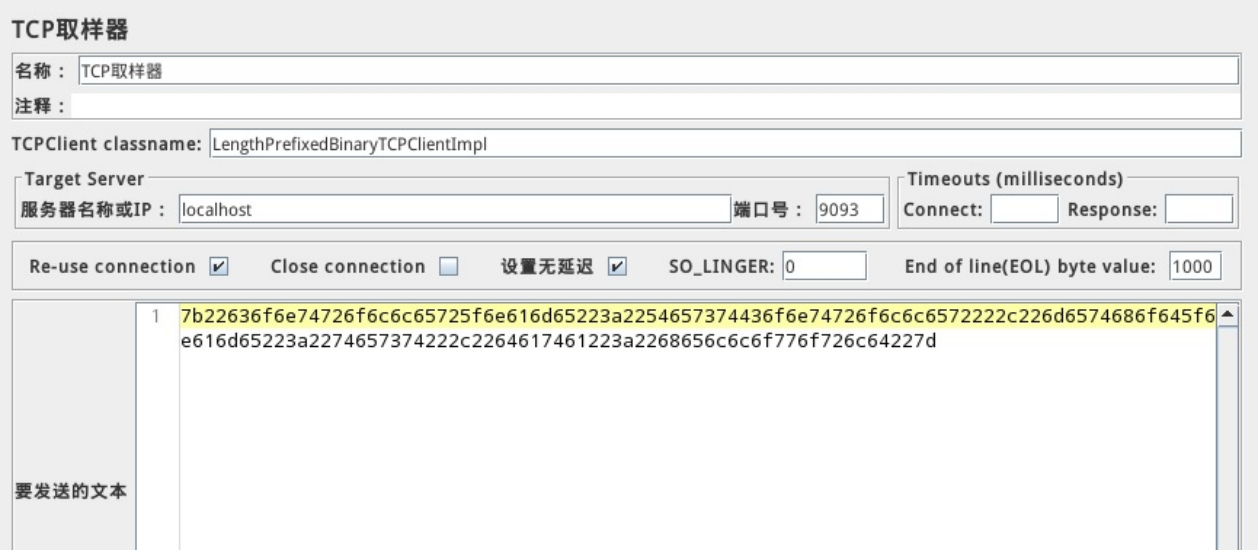
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received K...	Sent KB/sec
HTTP请求	100000	3	2	4	6	19	0	154	0.00%	21477.7/sec	3418.81	4215.83
总体	100000	3	2	4	6	19	0	154	0.00%	21477.7/sec	3418.81	4215.83

Tcp请求的JMeter聚合报告

Jmeter的Tcp测试需要改下服务器的默认配置，在(>1.7版本后)config中可以配置probuf_set.

```
$config['server']['probuf_set'] = [
    'open_length_check' => 1,
    'package_length_type' => 'n',
    'package_length_offset' => 0,       //第N个字节是包长度的值
    'package_body_offset' => 2,         //第几个字节开始计算长度
    'package_max_length' => 2000000,  //协议最大长度)
];
```

Tcp取样器的配置



由于配置的是二进制数据，所以发送的文本是HEX模式。

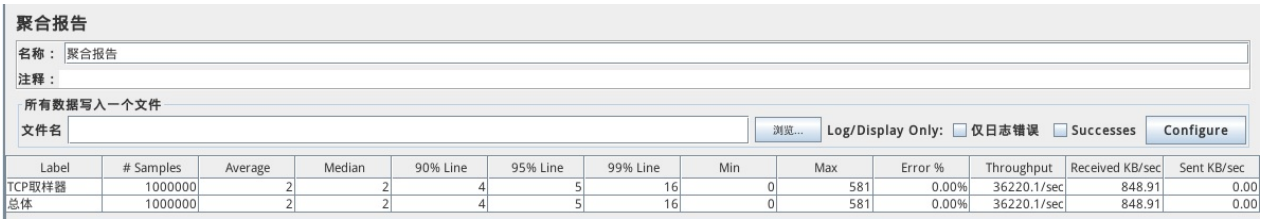
实际上发送的协议体内容为：

```
{"controller_name":"TestController","method_name":"test","data":
"helloworld"}
```

调用的方法是

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

聚合报告


聚合报告

名称： 聚合报告

注释：

所有数据写入一个文件

文件名 浏览... Log/Display Only: 仅日志错误 Successes Configure

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
TCP取样器	1000000	2	2	4	5	16	0	581	0.00%	36220.1/sec	848.91	0.00
总体	1000000	2	2	4	5	16	0	581	0.00%	36220.1/sec	848.91	0.00

附录：单元测试

仿照PHPUnit提供一个简易的单元测试框架

TestCase

这是基类，请继承。

方法

test开头的public方法才能作为测试用例，其余将被忽略。

setUpBeforeClass与tearDownAfterClass

setUpBeforeClass() 与 tearDownAfterClass() 模板方法将分别在测试用例类的第一个测试运行之前和测试用例类的最后一个测试运行之后调用

setUp与tearDown

测试类的每个测试方法都会运行一次 setUp() 和 tearDown() 模板方法

coroutineRequestHttpController

启动一个http的模拟访问。

```
        $testRequest = new TestRequest('/TestController/test');
        $testResponse = yield $this->coroutineRequestHttpControl
ler($testRequest);
        $this->assertEquals($testResponse->data, 'helloworld');
```

TestRequest中可以设置请求的一些方法。
testResponse为返回的数据。其中data为返回的值，其余见类成员。

coroutineRequestTcpController

启动一个tcp的模拟访问

```
        if ($this->config['server']['pack_tool'] != 'JsonPack') {

            $this->markTestSkipped('协议解包不是JsonPack');
        }
        $data = ['controller_name' => 'TestController', 'method_
name' => 'test', 'data' => 'helloWorld'];
        $reusult = yield $this->coroutineRequestTcpController($d
ata);
        $this->assertCount(2, $reusult);
```

$data传进去的是一个协议体

$result是返回的服务器具体操作步骤，详情可以自己打印。

特别注意这是模拟的方式，所以服务器不会产生任何的send操作，只是记录操作。

使用controller内提供的方法才能被记录，get_instance()的方法不会被记录，可能还会产生错误。

markTestSkipped

表示该测试被跳过。

@needTestTask

标注needTestTask

被标注的将会在测试的时候额外进行task同步测试。

@codeCoverageIgnore

标注codeCoverageIgnore

被标注的会在测试的时候被忽略

@depends

标注depends

对测试方法之间的显式依赖关系进行声明。

被标注的将产生依赖，和phpunit一样

```
public function testEmpty()
    {
        $stack = [];
        $this->assertEmpty($stack);

        return $stack;
    }

    /**
     * @depends testEmpty
     */
    public function testPush(array $stack)
    {
        array_push($stack, 'foo');
        $this->assertEquals('foo', $stack[count($stack)-1]);
        $this->assertNotEmpty($stack);

        return $stack;
    }

    /**
     * @depends testPush
     */
    public function testPop(array $stack)
    {
        $this->assertEquals('foo', array_pop($stack));
        $this->assertEmpty($stack);
    }
```

在上例中，第一个测试，testEmpty()，创建了一个新数组，并断言其为空。随后，此测试将此基境作为结果返回。第二个测试，testPush()，依赖于 testEmpty() ，并将所依赖的测试之结果作为参数传入。最后，testPop() 依赖于 testPush()。

@dataProvider

标注dataProvider

测试方法可以接受任意参数。这些参数由数据供给器方法提供。用 @dataProvider 标注来指定使用哪个数据供给器方法。

数据供给器方法必须声明为 public，其返回值要么是一个数组，其每个元素也是数组

例子：使用带有命名数据集的数据供给器

```
/**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertEquals($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            'adding zeros'  => [0, 0, 0],
            'zero plus one' => [0, 1, 1],
            'one plus zero' => [1, 0, 1],
            'one plus one'  => [1, 1, 3]
        ];
    }
```

如果测试同时从 @dataProvider 方法和一个或多个 @depends 测试接收数据，那么来自于数据供给器的参数将先于来自所依赖的测试的。来自于所依赖的测试的参数对于每个数据集都是一样的

例子: 在同一个测试中组合使用 @depends 和 @dataProvider

```
public function provider()
    {
        return [['provider1'], ['provider2']];
    }

    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

    public function testProducerSecond()
    {
        $this->assertTrue(true);
        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     * @dataProvider provider
     */
    public function testConsumer()
    {
        $this->assertEquals(
            ['provider1', 'first', 'second'],
            func_get_args()
        );
    }
```

各种简易断言

assertEquals

assertEmpty

assertNotEmpty

.....

更新日志

SwooleDistributed v1.7.9 更新(2017/03/16)

修复了一些bug。

SwooleDistributed v1.7.8 更新(2017/02/22)

主要更新内容：

三大组件Controller，Model，Task中增加了Context，是个数组结构。

Context就是上下文的意思，里面可以存储自定义的可序列化的信息，会传递给调用的Model和Task。

一次请求中Controller，Model，Task的Context是保持一致的。由于Model持有的是Context的引用，对Context的修改会影响前后，而Task获取的是Context的副本。

Context中默认会有个request_id表明该次请求的id。(主要应用于全链路监控系统，跟踪requestid经过了哪些server，方便定位log的位置，能在一定程度上缓解维护压力。)

注意：

Loader方法对loader->task进行的api的修改，增加了第二个参数，作用和loader->model方法一样，为了传递Context。

Model的initialization方法增加了一个传入参数$context。

Task的initialization方法增加了一个传入参数$context。

Controller的initialization方法会自动生成一个Context实例。

增加了获取Context的api：getContext();

SwooleDistributed v1.7.7 更新(2017/02/14)

主要更新内容：

增强SwooleTask

1.Task允许中断，比如耗时的Task可以通过中断取消运行。

2.获取机器上所有正在运行的Task信息。

SwooleDistributed v1.7.6 更新(2017/02/13)

主要更新内容：

1.连接池部分代码重构，支持多连接管理，比如用户可以创建新的redis/mysql连接池，框架默认还是使用配置中active标志的配置创建默认的redis和mysql连接池，所以更新对之前代码没有任何影响。传送门

2.提供了select方法用于协程返回值的选择，比如c1,c2,c3这3个协程只要一个返回值就立即执行其他的业务，这种场景就可以通过select实现。传送门

3.去除了一些不常用的回调模式

SwooleDistributed v1.7.5 更新(2017/02/08)

destroy拼写错误修复

loader加载可能引发内存泄漏及死循环的问题修复

controller，model不允许重写构造函数

增加initialization方法代替__construct

用户如果用到destory，可以全局进行拼写替换，destory替换为destroy，Destory替换为Destroy

SwooleDistributed v1.7.4 更新

http静态文件访问优化

http支持域名路由

HttpOutput,HttpInput 方法名驼峰化

1.7.3版本更新日志

1.修复了websocket，sendToAll的bug

2.HttpInPut增加了更多有用方便的函数

3.NormalRoute路由规则微调，首先先判断Controller有没有匹配，没有直接返回404，长连接直接回复异常，然后匹配Method，如果没有匹配则进入Controller->defalueMethod方法。

4.Controller中增加defalueMethod方法，提供了默认的处理逻辑

1.7.2版本更新日志

可以自定义协议头

SwooleDistributed v1.7 更新

文档已更新至1.7版本
最新文档

http://182.92.224.125/

注意事项

1、需要运行composer install 增加了依赖

2、增加新的命令'test'进行单元测试，详情见文档

3、如果使用过redis mget的请注意，为了和redis扩展返回值统一，1.7版本的mget返回值不包含对应的key了

功能更新

1、增加了单元测试通过test命令运行，见文档

2、异步redis和mysql与对应同步扩展调用api和回复的结构完全统一

3、提供了大量的单元测试用例

4、协程redis提供了一个新的调用方法getCoroutine()，使用ide的朋友可以看到代码提示。方便使用。

SwooleDistributed v1.6 年度重磅更新

4大模块Model，Controller，View，Task，Model处理异步简单事务，Task处理同步耗时事务，原本开发者需要在Model中使用异步客户端（异步redis，异步mysql），在Task中使用同步客户端（redis扩展，mysql-pdo扩展），代码风格完全不一致，无法重用。现在最新版本的SD框架将托管异步和同步客户端的调用，使用协程模式书写的代码在Model和Task中完全通用，并且Task和Model之间可以完成互相调用，代码100%可重用，采用协程代码风格开发者将可以忽略异步与同步的区别。通过task调用model的方法可以很容易将一个耗时任务优化到任务队列中而不需要更改任何代码。

SwooleDistributed v1.4 版本更新

1.增强群的功能，优化了群在redis中的存储结构，提供了丰富的群api

2.协程极小情况的报错的bug修复

3.增加开服初始化接口onOpenServiceInitialization

4.redis异步命令纠错，sadd现在也支持批量模式

5.mysql连接池断线重连优化

1.3.2更新日志

增加了同步方式的mysql，需要pdo扩展

pdo支持断线自动重连

语法构建器和异步一致

新增pdo开头的几个方法用于同步使用

task中使用同步mysql也变得容易了

1.3.1更新日志

1.修复了task中不能使用sendall，senduids的问题，现在可以使用了

2.由于效率问题，websocket和tcp暂时强制使用同一个pack。

3.Protobuf解析优化

4.redis和mysql一样加了一个active标签

SwooleDistributed v1.3

1.主要是关于http的，增加了www目录，优化了404页面，优化了路由策略，会自动匹配www目录下的文件，config添加了HTTP Content-type 对照表等。。。

2.代码结构稍微调整了下，app下增加了个AppServer继承SWD，启动文件入口改为指向AppServer

3.view的些许优化

4.swoole的gzip效率存在问题，替换了压缩的方法，qps显著提高

v1.2.2

1.格式化了上传的代码优化的导入，所以可能看起来很多的文件改变，其实并没有多少。

2.启动界面进行了优化，现在更清楚更简洁

3.配置增加了use_dispatch开关，默认为关，需要集群的请手动打开。

4.启动会询问是否清除redis上缓存的用户状态

5.提供了获取服务器在线用户数，判断用户是否在线的协程接口

开源协议

```
  Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, rep
roduction,
      and distribution as defined by Sections 1 through 9 of thi
s document.

      "Licensor" shall mean the copyright owner or entity author
ized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity a
nd all
      other entities that control, are controlled by, or are und
er common
      control with that entity. For the purposes of this definit
ion,
      "control" means (i) the power, direct or indirect, to caus
e the
      direction or management of such entity, whether by contrac
t or
      otherwise, or (ii) ownership of fifty percent (50%) or mor
e of the
      outstanding shares, or (iii) beneficial ownership of such
entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making mod
ifications,
```

```
      including but not limited to software source code, documen
tation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanica
l
      transformation or translation of a Source form, including
but
      not limited to compiled object code, generated documentati
on,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Sourc
e or
      Object form, made available under the License, as indicate
d by a
      copyright notice that is included in or attached to the wo
rk
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source
or Object
      form, that is based on (or derived from) the Work and for
which the
      editorial revisions, annotations, elaborations, or other m
odifications
      represent, as a whole, an original work of authorship. For
 the purposes
      of this License, Derivative Works shall not include works
that remain
      separable from, or merely link (or bind by name) to the in
terfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, includin
g
      the original version of the Work and any modifications or
additions
      to that Work or Derivative Works thereof, that is intentio
nally
      submitted to Licensor for inclusion in the Work by the cop
yright owner
```

```
      or by an individual or Legal Entity authorized to submit o
n behalf of
      the copyright owner. For the purposes of this definition,
"submitted"
      means any form of electronic, verbal, or written communica
tion sent
      to the Licensor or its representatives, including but not
limited to
      communication on electronic mailing lists, source code con
trol systems,
      and issue tracking systems that are managed by, or on beha
lf of, the
      Licensor for the purpose of discussing and improving the W
ork, but
      excluding communication that is conspicuously marked or ot
herwise
      designated in writing by the copyright owner as "Not a Con
tribution."

      "Contributor" shall mean Licensor and any individual or Le
gal Entity
      on behalf of whom a Contribution has been received by Lice
nsor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and condi
tions of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      copyright license to reproduce, prepare Derivative Works o
f,
      publicly display, publicly perform, sublicense, and distri
bute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditio
ns of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
```

```
able
      (except as stated in this section) patent license to make,
 have made,
      use, offer to sell, sell, import, and otherwise transfer t
he Work,
      where such license applies only to those patent claims lic
ensable
      by such Contributor that are necessarily infringed by thei
r
      Contribution(s) alone or by combination of their Contribut
ion(s)
      with the Work to which such Contribution(s) was submitted.
 If You
      institute patent litigation against any entity (including
a
      cross-claim or counterclaim in a lawsuit) alleging that th
e Work
      or a Contribution incorporated within the Work constitutes
 direct
      or contributory patent infringement, then any patent licen
ses
      granted to You under this License for that Work shall term
inate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of
 the
      Work or Derivative Works thereof in any medium, with or wi
thout
      modifications, and in Source or Object form, provided that
 You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent n
otices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative
Works
```

```
           that You distribute, all copyright, patent, trademark,
 and
           attribution notices from the Source form of the Work,
           excluding those notices that do not pertain to any par
t of
           the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of i
ts
           distribution, then any Derivative Works that You distr
ibute must
           include a readable copy of the attribution notices con
tained
           within such NOTICE file, excluding those notices that
do not
           pertain to any part of the Derivative Works, in at lea
st one
           of the following places: within a NOTICE text file dis
tributed
           as part of the Derivative Works; within the Source for
m or
           documentation, if provided along with the Derivative W
orks; or,
           within a display generated by the Derivative Works, if
 and
           wherever such third-party notices normally appear. The
 contents
           of the NOTICE file are for informational purposes only
 and
           do not modify the License. You may add Your own attrib
ution
           notices within Derivative Works that You distribute, a
longside
           or as an addendum to the NOTICE text from the Work, pr
ovided
           that such additional attribution notices cannot be con
strued
           as modifying the License.

      You may add Your own copyright statement to Your modificat
ions and
      may provide additional or different license terms and cond
```

```
itions
      for use, reproduction, or distribution of Your modificatio
ns, or
      for any such Derivative Works as a whole, provided Your us
e,
      reproduction, and distribution of the Work otherwise compl
ies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state o
therwise,
      any Contribution intentionally submitted for inclusion in
the Work
      by You to the Licensor shall be under the terms and condit
ions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede
or modify
      the terms of any separate license agreement you may have e
xecuted
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use
the trade
      names, trademarks, service marks, or product names of the
Licensor,
      except as required for reasonable and customary use in des
cribing the
      origin of the Work and reproducing the content of the NOTI
CE file.

   7. Disclaimer of Warranty. Unless required by applicable law
or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASI
S,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or
      implied, including, without limitation, any warranties or
conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FO
R A
```

```
      PARTICULAR PURPOSE. You are solely responsible for determi
ning the
      appropriateness of using or redistributing the Work and as
sume any
      risks associated with Your exercise of permissions under t
his License.

   8. Limitation of Liability. In no event and under no legal th
eory,
      whether in tort (including negligence), contract, or other
wise,
      unless required by applicable law (such as deliberate and
grossly
      negligent acts) or agreed to in writing, shall any Contrib
utor be
      liable to You for damages, including any direct, indirect,
 special,
      incidental, or consequential damages of any character aris
ing as a
      result of this License or out of the use or inability to u
se the
      Work (including but not limited to damages for loss of goo
dwill,
      work stoppage, computer failure or malfunction, or any and
 all
      other commercial damages or losses), even if such Contribu
tor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistri
buting
      the Work or Derivative Works thereof, You may choose to of
fer,
      and charge a fee for, acceptance of support, warranty, ind
emnity,
      or other liability obligations and/or rights consistent wi
th this
      License. However, in accepting such obligations, You may a
ct only
      on Your own behalf and on Your sole responsibility, not on
 behalf
      of any other Contributor, and only if You agree to indemni
```

```
fy,
      defend, and hold each Contributor harmless for any liabili
ty
      incurred by, or claims asserted against, such Contributor
by reason
      of your accepting any such warranty or additional liabilit
y.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the follo
wing
      boilerplate notice, with the fields enclosed by brackets "
[]"
      replaced with your own identifying information. (Don't inc
lude
      the brackets!)  The text should be enclosed in the appropr
iate
      comment syntax for the file format. We also recommend that
 a
      file or class name and description of purpose be included
on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright Jincheng.Zhang [tmtbe@163.com]

   Licensed under the Apache License, Version 2.0 (the "License"
);
   you may not use this file except in compliance with the Licen
se.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, so
ftware
   distributed under the License is distributed on an "AS IS" BA
SIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
```

```
or implied.
   See the License for the specific language governing permissio
ns and
   limitations under the License.
```

原理

开发适用范围

适合HttpRest开发，Tcp/Websocket长连接开发。

可以作为游戏服务器及API服务器。

本框架虽然包含了Web模块，模版模块，但目前缺乏具体项目支撑，所以并不建议在线上项目中作为web服务器。

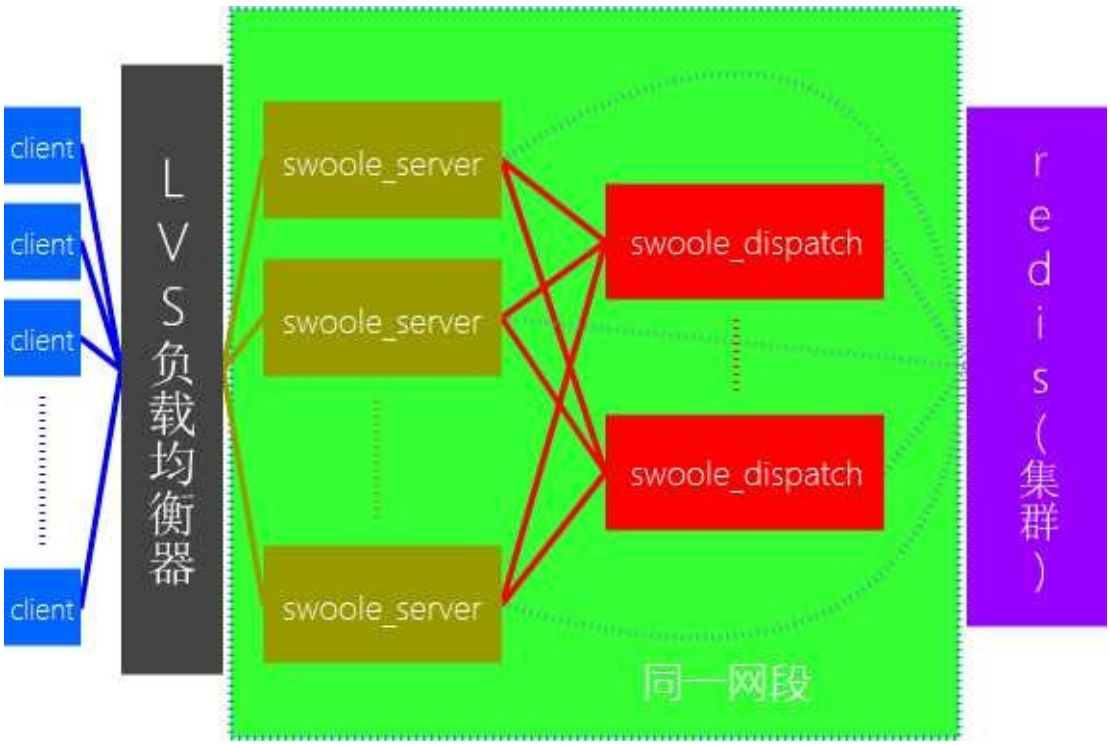
Server说明

Server是单台业务服务器，主要由一个Master，一个Manger，多个Worker和多个Task进程及一些辅助进程组成，是开发者重点开发维护的服务器，承载服务器业务和逻辑。

Dispatch说明

Dispatch是负责Server与Server之间通讯的分布式组件，一般情况下不需要额外的书写代码。

拓扑图



特点

并不是GateWay模式，连接由LVS负载均衡器维持，服务器之间的通讯由后端Dispatch模块支持，Redis或Redis集群作为数据共享。

Dispatch服务器因为不需要写Redis，所以可以单独配置只读的Redis提高性能。

不用太过于担心LVS负载的问题，通过使用VS/TUN，或者VS/DR技术一般情况下1台LVS可以支撑10台后端服务器，如果是部署在阿里云上的开发者可以直接选用阿里的负载均衡器。

由于分布式组件需要在一个网段所以阿里云上的部署需要购买VPC。

为何选择SD框架

PHP是一种被广泛应用的开源脚本语言，绝大多数开发者使用PHP做基于Web的应用程序，并且有了很多非常知名的Web框架。

传统的PHP应用程序基本上是在Apache等Web容器中运行的，浏览器与Web容器采用HTTP协议通信，然而在很多实际项目中HTTP协议无法满足我们的需求，尤其是在服务端和客户端要保持长连接，做实时双向通讯时，HTTP协议显得力不从心。例如即时IM通讯，游戏服务器通讯，与硬件传感器通讯等等，开发这些应用程序我们无法直接使用nginx/apache + PHP来实现。

Swoole扩展的出现使我们重新定义的PHP，我们通过Swoole扩展可以几乎做我们想做的任何事情，如基于Websocket的服务器、游戏服务器、移动通讯服务器、智能家居服务端、物联网服务、web服务器、RPC服务器等等。几乎任何基于TCP/UDP通讯的服务端都可以用PHP来开发。Swoole扩展使得开发者摆脱PHP只能用于Web开发的束缚，向更广阔的前景发展。

但是强大的Swoole并不那么容易使用，他作为一个扩展来说无一例外是强大的，优异的性能，丰富的Api，如何更容易的使用Swoole带来的性能和新特性，框架就要出来解决这一矛盾。

SwooleDistributed框架采用了MVC结构，他的一些灵感来自于CI框架，你们在使用的过程中会发现不少CI的影子，这使得我们使用过CI的开发者更容易上手。并且框架针对swoole的api进行了再次封装，在保证高性能的同时提供了高效率开发。

支持投递任务

针对swoole提供的Task进行了再次封装，Controller，Model中调用Task只需要一句话。

Task是swoole扩展中task的封装，在SwooleDistributed可以非常方便的进行异步（同步）任务投递。

定时器

定时器更容易使用，只需要更改配置随心所遇的调用Model或者Task的Api。

协程

不再需要复杂难看的回调，通过yield简化客户端代码。

PS：swoole2.0开始提供了协程，但是支持的并不完整比如task就没有实现协程模式，稳定性也没有1.9.x的稳定，框架目前使用的还是1.9.x版本。

托管同步异步

框架统一了同步与异步Redis/Mysql调用的API和返回格式，也就是说开发者不用自己选择使用同步还是异步，一切交给框架就行。

PS：这里带来的好处就是Task和Model的代码可以互相调用了。

单元测试

框架提供了一个类似PHPUNIT的单元测试模块。

连接池

异步Redis和异步Mysql的连接池不需要开发者维护，你们只需要使用就行。

SwooleDistributed借助Miner实现了Mysql的语法构建器。

Miner的用法：Miner

另外SwooleDistributed率先支持异步Mysql事务。

支持分布式系统

SwooleDistributed搭载dispatch组件实现分布式部署，俩者均是可以集群部署无单点故障。

单服务多协议

SwooleDistributed服务器通过配置不同的端口，可以同时处理tcp和http请求，比如开放http api实现给所有tcp的客户端广播消息这种需求在SwooleDistributed中实现起来非常简单。tcp和http请求通过自定义路由器可以指向同一个controller，再通过设置tcp，http方法的前缀可以分离tcp和http的处理方法。

支持全局广播或者向任意客户端推送数据

SwooleDistributed提供非常方便的API，可以全局广播数据、可以向某个群体广播数据、也可以向某个特定客户端推送数据。配合定时任务，也可以定时推送数据。

支持模板引擎

SwooleDistributed的模板引擎采用了plates。

plates的文档：plates

高并发，高性能，低配置

SwooleDistributed使用了swoole的扩展，内部的controller和model采用了对象池的模式，对象重用性能强悍。
SwooleDistributed和dispatch实现了内网自动发现，集群部署零配置。

支持服务平滑重启

当需要重启服务时（例如发布版本），我们不希望正在处理用户请求的进程被立刻终止，更不希望重启的那一刻导致客户端通讯失败。框架提供了平滑重启功能，能够保障服务平滑升级，不影响客户端的使用。

支持文件更新检测及自动加载

在开发过程中，我们希望在我们改动代码后能够立刻生效，以便查看结果。框架提供了文件检测及自动加载组件，只要文件有更新，框架会自动运行reload，以便加载新的文件，使之生效。

安装与配置

环境要求

运行所需环境

1、SwooleDistribute 要求运行在Linux环境下（centos、RedHat、Ubuntu、debian等）,也可以运行在mac os下（存在一定兼容性问题）。

2、安装有PHP-CLI(版本不小于7.0),并安装了Swoole1.9.x扩展。Redis，Mysql扩展也需要安装。

3、安装Composer，用于下载依赖。

详细说明

1、安装php7.0及其配套组件，php7.0-mysql，php7.0-dev

2、检查你的php扩展如果没有通过pecl安装扩展 redis，zip，mbstring，inotify，pdo_mysql

3、swoole官网下载最新的1.9.x版本的源码（不需要下载2.x版本）。

4、下载hiredis编译

```
make -j
sudo make install
sudo ldconfig
```

5、可以选择下载jemalloc编译

```
./configure
sudo make -j install
sudo ldconfig
```

6、开始编译swoole，如果跳过了第五步那么不需要在编译时加上--enable-jemalloc

```
phpize
./configure --enable-async-redis --enable-jemalloc
make clean
make -j
sudo make install
```

7、将所有的扩展添加到php.ini里

8、下载SD框架的源码

9、安装Composer

10、进入SD框架的目录，执行composer install开始下载依赖

这里建议中国开发者将composer换为中国镜像

```
composer config -g repo.packagist composer https://packagist.php
composer.com
composer install
```

11、如果需要protobuf支持，需要安装protobuf，建议安装2.6版本，更高版本我没有试过，但是2.6是一定可以的。

12、至此SD框架的环境就搭建完毕，可以修改服务器配置，试着启动服务器。

启动命令

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
```

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
```

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
```

服务器配置

config文件夹中的配置文件

config.php

服务器的配置

1、http_server http端口的设置

```
$config['http_server']['enable'] = true;
$config['http_server']['socket'] = '0.0.0.0';
$config['http_server']['port'] = 8081;
```

2、websocket的配置

这里注意opcode的设置，如果是文本传输需要变为WEBSOCKET_OPCODE_TEXT

```
/**
 * 是否启用websocket
 */
$config['websocket']['enable'] = true;
/*WEBSOCKET_OPCODE_TEXT = 0x1，UTF-8文本字符数据
WEBSOCKET_OPCODE_BINARY = 0x2，二进制数据*/
$config['websocket']['opcode'] = WEBSOCKET_OPCODE_BINARY;
```

3、Tcp的设置

```
/**
 * tcp设置
 */
$config['tcp']['enable'] = true;
$config['tcp']['socket'] = '0.0.0.0';
$config['tcp']['port'] = 9093;
```

4、服务器配置

dispatch_port：是连接dispatch服务器的端口，保持默认就好。

send_use_task_num：默认为20代表如果一次性向20个以上的客户端发送消息时会自动变成task去执行这一个操作。

pack_tool：序列化工具

route_tool：路由工具

set：swoole服务器配置，详情见swoole文档，其中worker_num代表启动的worker数，task_worker_num为启动的task数量，一般worker_num为cpu核数，dispatch_mode请不要修改默认为5，heartbeat_idle_time和heartbeat_check_interval为心跳配置。

注意这里的配置会被服务器启动类里的overrideSetConfig字段覆盖。

```
/**
 * 服务器设置
 */
$config['server']['dispatch_port'] = 9991;
$config['server']['send_use_task_num'] = 20;
$config['server']['log_path'] = '/../../';
$config['server']['log_max_files'] = 15;
$config['server']['log_level'] = \Monolog\Logger::DEBUG;
$config['server']['pack_tool'] = 'JsonPack';
$config['server']['route_tool'] = 'NormalRoute';
$config['server']['set'] = [
    '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,
    '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
];
```

注意start_swoole_server.php这里的overrideSetConfig。

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
$worker = new \app\AppServer();
$worker->overrideSetConfig = ['worker_num' => 4, 'task_worker_nu
m' => 2];
Server\SwooleServer::run();
```

5、协议头配置

这个功能在1.7版本以上才开放，修改以下字段可以改变协议头的定义。

其中open_length_check这个必须为1，不允许修改。

```
$config['server']['probuf_set'] = [
    'open_length_check' => 1,            //开启长度检测
    'package_length_type' => 'N',        //pack的方法
    'package_length_offset' => 0,        //第N个字节是包长度的值
    'package_body_offset' => 0,        //第几个字节开始计算长度
    'package_max_length' => 2000000,  //协议最大长度)
];
```

6、 dispatch_server

其中use_dispatch表示是否启动集群支持

port 是指发送udp广播的端口

password是指dispatch和server的验证密码，可以随意设置

set 参考swoole的set，保持默认就行也不需要修改

redis_slave 当dispatch和redis只读实例在同一台物理机时可以使用unixsock提高效率

```
$config['dispatch_server']['socket'] = '0.0.0.0';
   $config['dispatch_server']['port'] = 60000;
   $config['dispatch_server']['name'] = 'SwooleDispatch';
   $config['dispatch_server']['password'] = 'Hello Dsipatch';
   $config['dispatch_server']['set'] = [
    'reactor_num' => 2, //reactor thread num
    'worker_num' => 4,    //worker process num
    'backlog' => 128,   //listen backlog
    'open_tcp_nodelay' => 1,
    'dispatch_mode' => 3,
    'enable_reuse_port' => true,
   ];
   //主从redis提高读的速度
   //启动这个服务一定确保dispatch服务器上一定有一个redis只读服务器
   $config['dispatch_server']['redis_slave'] = ['unix:/var/run/redis/redis.sock',0];
   //是否启动集群的支持
   $config['use_dispatch'] = false;
```

注意：

1.7.6版本后$config['dispatch_server']['redis_slave']这个被删除，被放到了redis.php配置文件下。

7、 asyn_process_enable

```
//异步服务是否启动一个新进程（启动后异步效率会降低2倍，但维护连接池只有一个）

   $config['asyn_process_enable'] = false;
```

默认为false，每个进程都会维护个连接池，这样会导致连接的浪费但是效率最高。true将启动一个新的进程专门维护连接池，会有2次进程间通讯，效率只比同步方式高一点点。

8、 use_dispatch

默认为false工作在单机模式，如果需要集群的支持需要手动修改为true。

9、 auto_reload_enable

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

开启后src目录有更新会立即reload，需要安装inotify扩展

```
sudo pecl install inotify
```

10、coroution

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

11、asyn_process_enable

```
//异步服务是否启动一个新进程（启动后异步效率会降低2倍，但维护连接池只有一个）

$config['asyn_process_enable'] = false;
```

database.php

设置数据库连接
其中：

active 默认激活的mysql配置

asyn_max_count 是指异步连接池最多维护的连接数量

redis.php

设置redis连接
其中：

active 默认激活的redis配置

asyn_max_count 是指异步连接池最多维护的连接数量

注意：

如果没有密码下面这行需要整段注释掉

```
//$config['redis']['dispatch']['password']
```

1.7.6版本后，以下代码片段不能删除，代表的是dispatch服务器所使用的redis配置

```
/**
 * 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;
```

businessConfig.php

```
/**
 * tcp访问时方法的前缀
 */
$config['tcp']['method_prefix'] = '';
/**
 * http访问时方法的前缀
 */
$config['http']['method_prefix'] = 'http_';
/**
 * websocket访问时方法的前缀
 */
$config['websocket']['method_prefix'] = '';

//http服务器绑定的真实的域名或者ip:port，一定要填对,否则获取不到文件的绝对
路径
$config['http']['domain'] = 'http://localhost:8081';

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

//是否服务器启动时自动清除群组信息
$config['autoClearGroup'] = true;
```

设置前缀后访问controller的方法会根据设置加上前缀名。

比如http：//localhost/Test/test，对应的方法是Test controller中的http_test。

如果没有http_test方法将会返回404页面。

```
/**
 * 设置域名和Root之间的映射关系
 */

$config['http']['root'] = [
    'localhost' =>
        [
            'root' => 'localhost',
            'index' => 'index.html'
        ]
];
```

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。

timerTask.php

定时器任务的配置文件
值得一提的是，timerTask.php这个配置文件是支持热重载的，reload后会重新读取配置。

```
/**
 * 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）执行区间 [start_time,end_time) 格式： Y-m-d H:i:s 没有代表
一直执行
 * （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
 */
//dispatch发现广播，实现集群的实现
$config['timerTask'][] = [
    'task_name' => 'UdpDispatchTask',
    'method_name' => 'send',
    'interval_time' => '30'
];
```

task_name指的是Tasks中的类名。

model_name指的是Models中的类名。

其中代码中包含了2个定时器。

UdpDispatchTask：实现集群服务自发现，删除后自发现服务就不存在了。

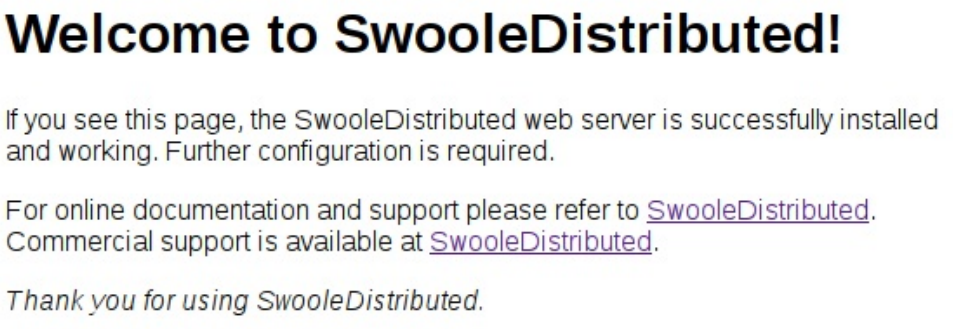
TestTask：测试，可以删除。

验证服务启动成功

如果开启了Http支持可以直接访问

```
http://loaclhost:8081
```

如果出现以下界面就代表服务启动成功了。



另外你也可以访问控制器方法

```
http://loaclhost:8081/TestController/test
```

如果是Tcp连接你将需要创建一个socket连接来测试，我们提供了一个测试文件start_swoole_client.php。

```php
$worker_num = 100;
$total_num = 100000;
$GLOBALS['package_length_type'] = 'N';
$GLOBALS['package_length_type_len'] = 4;
$GLOBALS['package_length_offset'] = 0;
$ips = ['127.0.0.1', '192.168.21.10'];
$GLOBALS['total_num'] = $total_num;
$GLOBALS['worker_num'] = $worker_num;
$GLOBALS['count'] = 0;
$GLOBALS['count_page'] = 0;
$GLOBALS['test_count'] = 0;
```

通过修改上面的配置，和你的服务器配置保持一致。其中ips是服务器列表，如果你只是测试单机那么只需要填写['127.0.0.1']即可。

```
function connect($cli)
{
    $cli->send(encode(json_pack('TestController', 'bind_uid', $c
li->i)));
    swoole_timer_after(1000, function () use ($cli) {
        $GLOBALS['start_time'] = getMillisecond();
        for ($i = 0; $i < $cli->total_num; $i++) {
            $cli->send(encode(json_pack('TestController', 'effic
iency_test', $GLOBALS['test_count'])));
            $GLOBALS['test_count']++;
        }
    });
}
```

这里的efficiency_test为测试集群性能，如果单机可以改为efficiency_test2。

开发流程

开发前必读

在你使用SD框架之前你需要了解

了解swoole扩展

在使用本框架前，请认真仔细的去了解下swoole扩展。

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

注意避免类和常量的重复定义

由于SD会缓存编译后的PHP文件，所以要避免多次require/include相同的类或者常量的定义文件。建议使用require_once/include_once加载文件。

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Controller的对象池模式

Controller是对象池模式，使用完后需要destroy，然后会被对象池回收。

注意不要使用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。

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，但是config文件夹中只有timerTask支持平滑重启。

目录结构

```
├── src // 代码文件夹
│   ├── test // 这里是单元测试用例目录
│   ├── app // 这里是开发者应用项目
│   │   ├── Controllers  // Controllers目录
│   │   ├── Models  // Models目录
│   │   ├── Tasks  // Tasks目录
│   │   ├── Views  // Views目录
│   │   ├── Pack  // 自定义TCP解包类目录
│   │   ├── Route // 自定义路由类目录
│   │   └── AppServer.php // app服务器
│   │
│   ├── www // 这里是放置静态文件的目录
│   │
│   ├── config // config目录
│   │   ├── config.php  // 服务器配置
│   │   ├── businessConfig.php   // 业务的一些配置
│   │   ├── datebase.php        // 数据库配置
│   │   ├── redis.php        // redis配置
│   │   ├── fileHeader.php    // 文件扩展名与http头的对照表
│   │   └── timerTask.php        // 定时任务配置
│   │
│   └── Server // 框架目录
│       ├── Controllers  // Controllers目录
│       ├── Models  // Models目录
│       ├── Tasks  // Tasks目录
│       │   ├── UdpDispatchTask.php  // 支持集群自发现服务的定时任务
│       │   └── UnitTestTask.php  // 支持Task的单元测试任务
│       ├── Views  // Views目录
│       │   └── error_404.php  // http访问时404页面模板
│       ├── Pack  // 自定义TCP解包类目录
│       │   ├── IPack.php  // 自定义pack的接口
│       │   ├── SerializePack.php  // SerializePack
│       │   ├── MsgPack.php  // MsgPack，需要安装msgpack扩展
│       │   └── JsonPack.php  // JsonPack
│       ├── Route // 自定义路由类目录
│       │   ├── IRoute.php  // 自定义route的接口
│       │   └── NormalRoute.php  // 提供的默认的route方案
```

```
│       ├── CoreBase // 框架核心代码
│       │   ├── Child.php  // 基类
│       │   ├── InotifyProcess //自动reload类
│       │   ├── Controller.php  // Controller基类
│       │   ├── ControllerFactory.php  // Controller工厂
│       │   ├── CoreBase.php  // core
│       │   ├── HttpInput.php  // Http输入
│       │   ├── HttpOutput.php  // Http输出
│       │   ├── Loader.php  // Loader
│       │   ├── Model.php  // Model基类
│       │   ├── ModelFactory.php  // Model工厂
│       │   ├── SwooleException.php  // swoole异常
│       │   ├── Coroutine.php  // 调度器
│       │   ├── CoroutineTask.php  // 调度器任务
│       │   ├── CoroutineNull //空
│       │   ├── ICoroutineBase.php  // 调度器任务接口
│       │   ├── TaskCoroutine.php //任务的协程
│       │   ├── Task.php  // Task基类
│       │   ├── TaskProxy.php  // Task代理
│       │   └── XssClean.php  // xss clean
│       ├── Client  // Client目录，目前存有HttpClient
│       │   ├── Client.php  // Client
│       │   ├── GetHttpClientCoroutine.php  // 获取httpclient的协
程
│       │   ├── HttpClient.php  // HttpClient
│       │   └── HttpClientRequestCoroutine.php  //http请求的协程
│       ├── DataBase // 数据库核心代码
│       │   ├── AsynPool.php  // 通用异步连接池
│       │   ├── AsynPoolManager.php  // 异步连接池管理器
│       │   ├── IAsynPool.php  // 异步连接池的接口
│       │   ├── Miner.php  // mysql语法生成器
│       │   ├── MySqlCoroutine //mysql的协程
│       │   ├── RedisCoroutine //redis的协程
│       │   ├── MysqlAsynPool.php  // mysql异步连接池
│       │   └── RedisAsynPool.php  // Redis异步连接池
│       ├── Test // 单元测试框架
│       │   ├── DocParser.php  // 文档解析器
│       │   ├── SwooleTestException.php  // 测试用异常
│       │   ├── TestCase.php  // 测试用例基类
│       │   ├── TestHttpCoroutine.php  // 用于获取httpController请
求结果
│       │   ├── TestModule.php //单元测试组件
```

```
│       │   ├── TestRequest.php //httpRequest
│       │   ├── TestResponse.php  // httpResponse
│       │   └── TestTcpCoroutine.php  // 用于获取tcpController请求
结果
│       ├── helpers   // 帮助函数库
│       │   └── Common.php  // Common工具函数
│       ├── SwooleDispatchClient.php   // dispatch服务
│       ├── SwooleDistributedServer.php   // SwooleDistributed服
务器
│       ├── SwooleHttpServer.php   // SwooleHttp服务器（基类）
│       ├── SwooleWebSocketServer.php   // SwooleWebSocket服务器
（基类）
│       ├── SwooleServer.php   // Swoole服务器（基类）
│       └── SwooleMarco.php    // 全局定义
│
├── composer.json //composer依赖管理json
├── LICENSE //开源协议
├── start_swoole_server.php // swoole_server启动脚本
├── start_swoole_dispatch.php // swoole_dispatch启动脚本
└── start_swoole_client.php // 测试工具
```

说明

开发者只需要关注app目录的文件即可，server目录为框架目录，开发者不要改动，这样方便后续框架升级。

start_swoole_server.php start_swoole_dispatch.php 分别是进程启动脚本，开发者一般不需要改动这俩个脚本。

开放规范

应用程序目录

我们提供了应用程序目录app，另外不要修改Server文件夹的文件，更好的方式是在app目录下通过继承来修改。

入口文件

SD中的应用程序需要一个入口文件，入口文件名没有要求，并且这个入口文件是以PHP Cli方式运行的。我们同样提供了一个模板start_swoole_server.php。

```
require_once __DIR__ . '/vendor/autoload.php';
$worker = new \app\AppServer();
$worker->overrideSetConfig = ['worker_num' => 4, 'task_worker_nu
m' => 2];
Server\SwooleServer::run();
```

需要遵循的psr-4代码规范

1、类采用首字母大写的驼峰式命名，类文件名称必须与文件内部类名相同，以便自动加载。

2、使用命名空间，命名空间名字与目录路径对应。

基本流程

简单的HTTP REST服务器

1、需要在config配置中启动http的支持，并修改成你想用的端口。

2、app/AppServer.php中可以做服务器的初始化工作。

3、app/Controllers目录中新建一个AppController.php继承Controller

4、可以写一些逻辑了

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

    /**
     * http测试
     */
    public function http_test()
    {
        $this->AppModel = $this->loader->model('AppModel', $this)
;
        $this->http_output->end($this->AppModel->test());
    }

    public function http_test_task()
    {
        $AppTask = $this->loader->task('AppTask');
        $AppTask->testTask();
        $AppTask->startTask(function ($serv, $task_id, $data) {
            $this->http_output->end($data);
        });
    }
}
```

5、我们启动服务器，访问http://localhost:8081/AppController/test

简单的TCP服务器

1、需要在config配置中启动tcp的支持，并修改成你想用的端口。

2、app/AppServer.php中可以做服务器的初始化工作。

3、app/Controllers目录中新建一个AppController.php继承Controller

4、可以写一些逻辑了

```
class AppController extends Controller
{
    public function test()
    {
        $this->send('helloworld');
    }
}
```

初识协程

迭代器

迭代生成器

(迭代)生成器也是一个函数,不同的是这个函数的返回值是依次返回,而不是只返回一个单独的值.或者,换句话说,生成器使你能更方便的实现了迭代器接口.下面通过实现一个xrange函数来简单说明：

```
<?php
function xrange($start, $end, $step = 1) {
    for ($i = $start; $i <= $end; $i += $step) {
        yield $i;
    }
}

foreach (xrange(1, 1000000) as $num) {
    echo $num, "\n";
}
```

上面这个xrange()函数提供了和PHP的内建函数range()一样的功能.但是不同的是range()函数返回的是一个包含值从1到100万0的数组(注：请查看手册). 而xrange()函数返回的是依次输出这些值的一个迭代器, 而不会真正以数组形式返回.

这种方法的优点是显而易见的.它可以让你在处理大数据集合的时候不用一次性的加载到内存中.甚至你可以处理无限大的数据流.

当然,也可以不同通过生成器来实现这个功能,而是可以通过继承Iterator接口实现.但通过使用生成器实现起来会更方便,不用再去实现iterator接口中的5个方法了.

生成器为可中断的函数

要从生成器认识协程, 理解它内部是如何工作是非常重要的: 生成器是一种可中断的函数, 在它里面的yield构成了中断点.

还是看上面的例子, 调用xrange(1,1000000)的时候, xrange()函数里代码其实并没有真正地运行. 它只是返回了一个迭代器：

```php
<?php
$range = xrange(1, 1000000);
var_dump($range); // object(Generator)#1
var_dump($range instanceof Iterator); // bool(true)
?>
```

这也解释了为什么xrange叫做迭代生成器, 因为它返回一个迭代器, 而这个迭代器实现了Iterator接口.

调用迭代器的方法一次, 其中的代码运行一次.例如, 如果你调用$range->rewind(), 那么xrange()里的代码就会运行到控制流第一次出现yield的地方. 而函数内传递给yield语句的返回值可以通过$range->current()获取.

为了继续执行生成器中yield后的代码, 你就需要调用$range->next()方法. 这将再次启动生成器, 直到下一次yield语句出现. 因此,连续调用next()和current()方法, 你就能从生成器里获得所有的值, 直到再没有yield语句出现.

对xrange()来说, 这种情形出现在$i超过$end时. 在这中情况下, 控制流将到达函数的终点,因此将不执行任何代码.一旦这种情况发生,vaild()方法将返回假, 这时迭代结束.

调度器

使用协程实现多任务协作，我们要解决的问题是你想并发地运行多任务(或者“程序”)，不过我们都知道CPU在一个时刻只能运行一个任务（不考虑多核的情况），因此处理器需要在不同的任务之间进行切换,而且总是让每个任务运行“一小会儿”。

多任务协作这个术语中的“协作”很好的说明了如何进行这种切换的：它要求当前正在运行的任务自动把控制传回给调度器,这样就可以运行其他任务了. 这与“抢占”多任务相反, 抢占多任务是这样的：调度器可以中断运行了一段时间的任务, 不管它喜欢还是不喜欢. 协作多任务在Windows的早期版本(windows95)和Mac OS中有使用, 不过它们后来都切换到使用抢先多任务了. 理由相当明确：如果你依靠程序自动交出控制的话, 那么一些恶意的程序将很容易占用整个CPU, 不与其他任务共享。

现在你应当明白协程和任务调度之间的关系：yield指令提供了任务中断自身的一种方法, 然后把控制交回给任务调度器. 因此协程可以运行多个其他任务. 更进一步来说, yield还可以用来在任务和调度器之间进行通信。

SD框架的任务调度器是基于定时器的，默认每过1ms时间会调度所有任务的运行，使得所有任务进行下一步的操作，比如下面的代码段：

```
while(true)
{
    echo 1;
}
```

这段代码在swoole中其实是不允许出现的，因为死循环会导致主进程的堵塞，从而swoole无法处理其他的逻辑。但是如果我们加上yield，并且这段代码是在调度器中被调度的话（这段代码作为协程运行），调度器将会每隔1ms调度一次任务，然后释放cpu，等待下一个1ms。

```
while(true)
{
    echo 1;
    yiled;
}
```

我们会发现服务器不停的在打印，但是却不会被堵塞。

协程

在SD框架中协程基本无处不在，在任何我们支持协程的地方你都可以利用协程的便利性。

Task，Model，Controller，及AppServer的回调操作中均支持协程。

而协程方式的API在SD框架中有着明显的标志，它们均含有Coroutine字段。

使用协程最明显的改善就是我们可以以同步的方式来书写异步的代码。

AppServer特定函数内支持协程写法

onOpenServiceInitialization 开服初始化

onUidCloseClear 当一个绑定uid的连接close后的清理

Model

所有的函数内部都支持协程

Controller

所有的函数内部都支持协程

Task

所有的函数内部都支持协程

虽然Task是同步的模块，但它依旧支持协程的写法，框架会自动转换为同步的方式执行。

使用协程的优势

Swoole是高性能异步框架，正因为异步所以高性能，但是异步也有异步的不好，写逻辑代码有时候就非常的不方便，需要多层嵌套回调，弄得代码的可读性很差维护起来非常的不方便，那么如何解决这个弊端呢？那就是使用协程。

SwooleDistributed框架提供了一套基于yield关键字的协程写法。

回调风格和协程风格的区别

举例说明：

回调风格：

```
    /**
     * mysql 测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function mysql_test()
    {
        $this->mysql_pool->dbQueryBuilder->select('*')->from('ac
count')->where('sex', 1);
        $this->mysql_pool->query(function ($result) {
            print_r($result);
        });
        $this->destroy();
    }
```

协程风格：

```
    /**
     * mysql 测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function mysql_test()
    {
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('sex', 1)->coroutineSend();
        $result = yield $mySqlCoroutine;
        print_r($result);
        $this->destroy();
    }
```

上述代码还只是基础，想想看多次请求并且依赖的情况 回调风格：

```
    public function test($callback)
    {
        $this->redis_pool->get('test',function ($uid)use($callba
ck){
            $this->mysql_pool->dbQueryBuilder->select('*')->from(
'account')->where('uid', $uid);
            $this->mysql_pool->query(function ($result)use($call
back) {
                call_user_func($callback,$result);
            });
        });
    }
```

协程风格：

```
    public function test()
    {
        $redisCoroutine = $this->redis_pool->getCoroutine()->get(
'test');
        $uid = yield $redisCoroutine;
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('uid', $uid)->coroutineSend();
        $result = yield $mySqlCoroutine;
        return $result;
    }
```

上面的代码是在一个model中，按照以往的回调风格，controller调用这个model还需要传进去一个回调函数才能获取到值（我觉得这么做的人一定会疯），而协程风格你只需要按部就班的return结果就行了。和同步的代码唯一的区别就是多了一个yield关键字。

好了我们再看调用这个model的controller怎么写。
回调风格：

```
    /**
     * 非协程测试
     */
    public function http_testNOCoroutine()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $this->testModel->test(function($result){
           $this->http_output->end($result);
        });
    }
```

协程风格：

```
    /**
     * 协程测试
     */
    public function http_testCoroutine()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $result = yield $this->testModel->test();
        $this->http_output->end($result);
    }
```

同样只是要多加一个yield关键字。是不是很方便～，注意只有这个model的方法内使用了yield，控制器调用的时候才需要加yield关键字。普通的方法是不需要的，当然你加了也不会报错。

协程中的异常？

swooleDistributed框架已经将协程的使用变得很方便了，至于异常和平常的写法一毛一样。

```
    public function test_exceptionII()
    {
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('uid', 10303)->coroutineSend();
        $result = yield $mySqlCoroutine;
        throw new \Exception('test');
    }
```

无论你是在model中还是在controller中，还是在model的model中。。。。
总之throw出来就行，上一层使用try可以捕获到错误。
这里还有个小小的体验优化,当报错一直到controller层的时候，会被controller的onExceptionHandle捕获到，你可以重写这个函数实现异常情况下的客户端回复。

协程的嵌套

随便嵌套。enjoy yourself。

协程的调度顺序

调节yield的位置即可调节接收的顺序。

```
    $redisCoroutine = $this->redis_pool->getCoroutine()->get('te
st');
    $uid = yield $redisCoroutine;
    $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select(
'*')->from('account')->where('uid', $uid)->coroutineSend();
    $result = yield $mySqlCoroutine;
```

上面的代码调度顺序是redis_send->redis_rev->mysql_send->mysql_rev。

```
    $redisCoroutine = $this->redis_pool->getCoroutine()->get('te
st');
    $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select(
'*')->from('account')->where(123, $uid)->coroutineSend();
    $uid = yield $redisCoroutine;
    $result = yield $mySqlCoroutine;
```

上面的代码调度顺序是redis_send->mysql_send->redis_rev->mysql_rev。

通过协程的方法屏蔽异步同步的区别

同步

sd框架中Task就是一个典型的同步案例。task中不允许调用异步的api。

异步

sd框架中除了task基本都是异步的，异步最好使用异步api达到更高的效率。

如何选择

在1.6版本之前，task中如果使用mysql和redis都必须调用同步的客户端，而且同步客户端和异步客户端的调用方法以及返回结构都不一样，这样model和task的代码完全无法重用，也更不可能通过task去调用model。

无需选择

1.6版本后实现了框架进行异步和同步的选择，1.7版本对于这种新的方式进行了优化和完善。 通过协程的方式写的代码可以同时在modle和task中运行。也不用关心同步和异步的写法不同，他们的调用方式和返回值都保持绝对的一致。

```
 $value = yield $this->redis_pool->getCoroutine()->setex('test', 10, 'testRedis');
```

比如这段代码在model和task中均能被正确的执行。

同时在task中也可以调用model的方法。

```
$testModel = $this->loader->model('TestModel', $this);
$result = yield $testModel->test_task();
```

至此难为程序员的异步和同步的区别现在被完美的解决了。

Select多路选择器

1.7.6新增功能

多路选择器的功能介绍

比如我们有ABCD四个任务，我们希望这4个任务并发的执行，最后谁返回的结果最快我们便选择那个结果，不等待其余任务的结果返回，不堵塞。

下面提供一个操作实例

```
/**
 * select方法测试
 * @return \Generator
 */
public function http_test_select()
{
    yield $this->redis_pool->getCoroutine()->set('test', 1);
    $c1 = $this->redis_pool->getCoroutine()->get('test');
    $c2 = $this->redis_pool->getCoroutine()->get('test1');
    $result = yield SelectCoroutine::Select(function ($result) {
        if ($result != null) {
            return true;
        }
        return false;
    }, $c2, $c1);
    $this->http_output->end($result);
}
```

Select函数原型

```
function Select($matchFunc, CoroutineBase ...$coroutines)
```

SelectCoroutine::Select函数可以接受n个参数，第一个参数是匹配函数用于筛选，如果结果满足便返回true否则返回false，第二到n个参数为CoroutineBase。

上述代码将在c1,c2中选择一个最快速的非null的值。得到需要结果后不会继续等待其他的结果。

需要注意的是每个协程都有拥有超时限制，这里SelectCoroutine在超时时间内没有获得符合条件的结果将会报超时错误。

封装器与路由器

路由帮助你实现url的解析，或者是协议解析后的跳转。

封装器作为一个序列化和反序列化的工具。（仅仅TCP，WebSocket需要）

SD框架默认提供了一个简单路由。路由的设置在config文件夹中的config.php中。

```
$config['server']['pack_tool'] = 'JsonPack';
$config['server']['route_tool'] = 'NormalRoute';
```

NormalRoute指向的是\Server\Route\NormalRoute类。当然他同样遵循优先寻找\app\Route\NormalRoute类，如果存在将优先使用。

JsonPack指向的是\Server\Pack\JsonPack类。当然他同样遵循优先寻找\app\Pack\JsonPack类，如果存在将优先使用。

封装器

Tcp基本都是自定义协议。这里涉及到协议的pack与unpack。 在默认的配置中pack工具使用的是

```
$config['server']['pack_tool'] = 'JsonPack';
```

对应的类为\Server\Pack\JsonPack，这个类是通过json作为协议的载体。 同样在businessConfig中也存在一个前缀定义默认为空。

```
$config['tcp']['method_prefix'] = '';
```

服务器接收到客户端的消息，

第一步将进行去掉协议头的操作获得协议体，客户端发起的一个完整的请求结构如下：

```
协议头                          协议体
N（4字节代表协议长度）          Body（各种载体封装的数据）
```

1.7.1之前的版本协议头的定义是固定死的，不允许修改。

1.7.2开始在config.php配置中允许修改协议头。

```
$config['server']['probuf_set'] = [
    'open_length_check' => 1,
    'package_length_type' => 'n',
    'package_length_offset' => 0,       //第N个字节是包长度的值
    'package_body_offset' => 0,       //第几个字节开始计算长度
    'package_max_length' => 2000000,  //协议最大长度)
];
```

第二步JsonPack类通过unPack函数解析Body，

第三步将转换后获得的client_data（是个类不是数组）传递给NormalRoute。

这个client_data需要满足2个基本字段。

1.$this->client_data->controller_name 获取控制器名称

2.$this->client_data->method_name 获取方法名称

例子：

客户端原始数据如下：

```
function encode($buffer)
{
    $total_length = 4 + strlen($buffer);
    return pack('N', $total_length) . $buffer;
}
$data['controller_name'] = 'TestController';
$data['method_name'] = 'test';
$data['data1'] = '123';
$data['data2'] = '456';
$data['data3'] = ['a','b'];
$jsonData = json_encode($pdata, JSON_UNESCAPED_UNICODE);
$sendData = encode($jsonData);
```

$sendData为客户端发送给服务器的数据，包含了协议头和协议体。

服务器收到消息后进行unEncode->unPack->route,最后访问到TestController控制器的test方法。

```
/**
 * tcp的测试
 */
public function test()
{
    echo $this->client_data->controller_name;
    echo $this->client_data->controller_name;
    echo $this->client_data->data1;
    echo $this->client_data->data2;
    echo $this->client_data->data3;
    $this->send($this->client_data);
}
```

控制器获取的client_data作为一个类。

自定义封装器

封装器只在Tcp或者WebSocket需要被设置。

封装器需要实现2个方法一个是pack序列化的方法，一个是unPack反序列化的方法。

```
interface IPack
{
    function pack($data);

    function unPack($data);
}
```

路由器

Http默认路由

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，它将和businessConfig中的

```
$config['http']['method_prefix'] = 'http_';
```

合并成访问的方法名，默认前缀名为'http_'，可以通过修改配置自己设置。

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);
    }
}
```

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();
}
```

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

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 $responseMes
sage)
    {
       //这里的$responseMessage就是getParams()获取到的对象
    }
}
```

AppServer

这个类是继承SwooleDistributedServer，提供给开发者维护的。

clearState

开服前的清理操作，只支持同步的写法。

例子：

```
    public function clearState()
    {
        print("是否清除Redis上的用户状态信息(y/n)？");
        $clear_redis = shell_read();
        if (strtolower($clear_redis) == 'y') {
            echo "[初始化] 清除Redis上用户状态。\n";
            $this->getRedis()->del(SwooleMarco::redis_uid_usid_h
ash_name);
            $this->getRedis()->close();
            unset($this->redis_client);
        }
    }
```

shell_read是在Common.php中定义的方法。用户获取用户的输入。

onOpenServiceInitialization

开服初始化的回调，开发者可在这里写开服的处理，这里保证无论多少进程只执行一次。

例子：

```
    /**
     * 开服初始化(支持协程)
     * @return mixed
     */
    public function onOpenServiceInitialization()
    {
        parent::onOpenServiceInitialization();
        yield $this->redis_pool->getCoroutine()->del(["account",
"uids"]);
    }
```

onUidCloseClear

当一个绑定uid的连接close后的清理 支持协程

例子：

```
    /**
     * 当一个绑定uid的连接close后的清理
     * 支持协程
     * @param $uid
     */
    public function onUidCloseClear($uid)
    {
        yield $this->redis_pool->getCoroutine()->rpush('logout', $uid);
    }
```

initAsynPools

这个方法在1.7.6版本以后才有

用于初始化连接池，可以增加更多的连接池。

```
    /**
     * 这里可以进行额外的异步连接池，比如另一组redis/mysql连接
     * @return array
     */
    public function initAsynPools()
    {
        parent::initAsynPools();
        $this->addAsynPool('redis2', new RedisAsynPool($this->co
nfig, 'test2'));
    }
```

如上面代码所展示我们增加了一个名为redis2的连接池。

RedisAsynPool的第一个参数传递我们的配置文件这里是$this->config，第二个参数传递的是配置中的目标名称这里是test2。

```
#redis.php文件
/**
 * 本地环境
 */
$config['redis']['test']['ip'] = '192.168.21.10';
$config['redis']['test']['port'] = 6379;
$config['redis']['test']['select'] = 1;
$config['redis']['test']['password'] = '123456';
$config['redis']['asyn_max_count'] = 10;

/**
 * 本地环境2
 */
$config['redis']['test2']['ip'] = '192.168.21.10';
$config['redis']['test2']['port'] = 6379;
$config['redis']['test2']['select'] = 2;
$config['redis']['test2']['password'] = '123456';
$config['redis']['asyn_max_count'] = 10;
```

这样一来名为redis2的连接池所访问的便是本地环境2所代表的redis连接了。

关于连接池还有2个方法

*addAsynPool 参考9.25章节

*getAsynPool 参考9.26章节

SwooleDistributedServer

SwooleDistributedServer，非常重要的功能类

get_instance

获取SwooleDistributedServer的实例。

Common.php中也有get_instance，效果和这个等同。

例子:

```
    public function __construct()
    {
        if (!empty(get_instance())) {
            $this->loader = get_instance()->loader;
            $this->logger = get_instance()->log;
            $this->server = get_instance()->server;
            $this->config = get_instance()->config;
            $this->pack = get_instance()->pack;
        }
    }
```

getRedis

获取Redis的同步实例。

注意，在1.7版本后我们推荐统一使用协程的方式调用Redis，Mysql。使用协程的API调用可以自动分配异步客户端或者是同步客户端。

协程的例子：

```
    public function testRedisDecrBy()
    {
        yield $this->redis_pool->getCoroutine()->set('key1', 10);

        yield $this->redis_pool->getCoroutine()->decrBy('key1',
10);
        $value = yield $this->redis_pool->getCoroutine()->get('k
ey1');
    }
```

getMysql

获取Mysql的同步实例。

注意，在1.7版本后我们推荐统一使用协程的方式调用Redis，Mysql。使用协程的API调用可以自动分配异步客户端或者是同步客户端。

协程的例子：

```
    /**
     * mysql Insert 命令
     * @return \Generator
     */
    public function testMysqlInsert()
    {
        $value = yield $this->mysql_pool->dbQueryBuilder->insert('MysqlTest')
            ->option('HIGH_PRIORITY')
            ->set('firstname', 'White')
            ->set('lastname', 'Cat')
            ->set('age', '25')
            ->set('townid', '10000')->coroutineSend();
    }
```

sendToUid

TCP/WebSocket可以使用。

向一个Uid绑定的客户端发送消息。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 向uid发送消息
     * @param $uid
     * @param $data
     * @param $fromDispatch
     */
    public function sendToUid($uid, $data, $fromDispatch = false)
```

例子：

```
    /**
     * 效率测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function efficiency_test()
    {
        $data = $this->client_data->data;
        $this->sendToUid(mt_rand(1, 100), $data);
    }
```

集群模式首先在本地服务器寻找匹配uid的客户端连接，如果没有找到则通过Dispatch服务器进行定位，找到并通知对应的服务器进行处理。

sendToUids

TCP/WebSocket可以使用。

向多个Uid绑定的客户端发送消息。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 批量发送消息
     * @param $uids
     * @param $data
     * @param $fromDispatch
     */
    public function sendToUids($uids, $data, $fromDispatch = fal
se)
```

例子：

```
   /**
     * 效率测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function efficiency_test()
    {
        $data = $this->client_data->data;
        $this->sendToUids([1,2,3], $data);
    }
```

集群模式首先在本地服务器寻找匹配uid的客户端连接，如果没有找到则通过Dispatch服务器进行定位，找到并通知对应的服务器进行处理。

kickUid

TCP/WebSocket可以使用。

强制Uid匹配的客户端下线。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 踢用户下线
     * @param $uid
     * @param bool $fromDispatch
     */
    public function kickUid($uid, $fromDispatch = false)
```

例子：

```
   /**
     * 效率测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function efficiency_test()
    {
        $this->kickUid(1000);
    }
```

集群模式首先在本地服务器寻找匹配uid的客户端连接，如果没有找到则通过Dispatch服务器进行定位，找到并通知对应的服务器进行处理。

delAllGroups

删除所有的群

bindUid

TCP/WebSocket可以使用。

为客户端连接绑定一个Uid

函数原型：

```
    /**
     * 将fd绑定到uid,uid不能为0
     * @param $fd
     * @param $uid
     * @param bool $isKick 是否踢掉uid上一个的链接
     */
    public function bindUid($fd, $uid, $isKick = true)
```

例子：

```
/**
* 效率测试
* @throws \Server\CoreBase\SwooleException
*/
public function efficiency_test()
{
    $this->bindUid($this->fd,1000);
}
```

注意uid不能为0。

默认$isKick为true，如果这个uid已经绑定了一个客户端连接，那么这次的绑定会强制上个客户端下线。

unBindUid

TCP/WebSocket可以使用。

为客户端连接解绑Uid

函数原型：

```
    /**
     * 解绑uid，链接断开自动解绑
     * @param $uid
     */
    public function unBindUid($uid)
```

例子：

```
/**
* 效率测试
* @throws \Server\CoreBase\SwooleException
*/
public function efficiency_test()
{
    $this->unBindUid(1000);
}
```

客户端连接断开后会自动执行unBindUid操作

coroutineUidIsOnline

TCP/WebSocket可以使用。

协程的方式获取该Uid客户端是否在线。

函数原型：

```
    /**
     * uid是否在线(协程)
     * @param $uid
     * @return RedisCoroutine
     * @throws SwooleException
     */
    public function coroutineUidIsOnline($uid)
```

协程例子：

```
    public function isOnline()
    {
        $value = yield get_instance()->coroutineUidIsOnline(1000)
;
        var_dump($value);
        $this->destroy();
    }
```

coroutineCountOnline

TCP/WebSocket可以使用。

协程的方式获取有多少客户端在线。

在集群模式下同样可以使用。

协程例子：

```
    public function isOnline()
    {
        $value = yield get_instance()->coroutineCountOnline();
        var_dump($value);
        $this->destroy();
    }
```

coroutineGetAllGroups

TCP/WebSocket可以使用。

协程的方式获取有多少客户端在线。

在集群模式下同样可以使用。

协程例子：

```
    public function getAllGroups()
    {
        $value = yield get_instance()->coroutineGetAllGroups();
        var_dump($value);
        $this->destroy();
    }
```

addToGroup

TCP/WebSocket可以使用。

将客户端连接添加到一个组中。

在集群模式下同样可以使用。

函数原型：

```
/**
 * 添加到群(可以支持批量,实际是否支持根据sdk版本测试)
 * @param $uid int | array
 * @param $group_id int
 */
public function addToGroup($uid, $group_id)
```

例子：

```
public function addToGroup()
{
    get_instance()->addToGroup(1000,1000000);
    $this->destroy();
}

public function addToGroupII()
{
    get_instance()->addToGroup([1000,1001],1000000);
    $this->destroy();
}
```

removeFromGroup

TCP/WebSocket可以使用。

将客户端连接从组中移除。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 从群里移除(可以支持批量,实际是否支持根据sdk版本测试)
     * @param $uid int | array
     * @param $group_id
     */
    public function removeFromGroup($uid, $group_id)
```

例子：

```
    public function removeFromGroup()
    {
        get_instance()->removeFromGroup(1000,1000000);
        $this->destroy();
    }

    public function removeFromGroup()
    {
        get_instance()->removeFromGroup([1000,1001],1000000);
        $this->destroy();
    }
```

delGroup

TCP/WebSocket可以使用。

删除这个组。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 删除群
     * @param $group_id
     */
    public function delGroup($group_id)
```

例子：

```
    public function delGroup()
    {
        get_instance()->delGroup(1000000);
        $this->destroy();
    }
```

coroutineGetGroupCount

TCP/WebSocket可以使用。

协程的方式获取该群的人数。

函数原型：

```
    /**
     * 获取群的人数（协程）
     * @param $group_id
     * @return RedisCoroutine
     * @throws SwooleException
     */
    public function coroutineGetGroupCount($group_id)
```

协程例子：

```
    public function getGroupCount()
    {
        $value = yield get_instance()->coroutineGetGroupCount(10000);
        var_dump($value);
        $this->destroy();
    }
```

coroutineGetGroupUids

TCP/WebSocket可以使用。

协程的方式获取该群的人数。

函数原型：

```
    /**
     * 获取群成员uids (协程)
     * @param $group_id
     * @return RedisCoroutine
     * @throws SwooleException
     */
    public function coroutineGetGroupUids($group_id)
```

协程例子：

```
    public function getGroupUids()
    {
        $value = yield get_instance()->coroutineGetGroupUids(100
00);
        var_dump($value);
        $this->destroy();
    }
```

sendToAll

TCP/WebSocket可以使用。

全服务器广播消息。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 广播
     * @param $data
     */
    public function sendToAll($data)
```

例子：

```
   /**
     * 效率测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function efficiency_test()
    {
        $data = $this->client_data->data;
        $this->sendToAll($data);
    }
```

集群模式会告诉Dispatch进行广播。

sendToGroup

TCP/WebSocket可以使用。

向Group组广播消息。

在集群模式下同样可以使用。

函数原型：

```
    /**
     * 发送给群
     * @param $groupId
     * @param $data
     */
    public function sendToGroup($groupId, $data)
```

例子：

```
   /**
     * 效率测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function efficiency_test()
    {
        $data = $this->client_data->data;
        $this->sendToGroup(10000，$data);
    }
```

集群模式会告诉Dispatch进行群广播。

setTemplateEngine

设置模板引擎

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

isWebSocket

判断这个fd是不是一个WebSocket连接，用于区分TCP和WebSocket

在SD框架中WebSocket的方法和TCP是完全一致的，所以当你需要判断这个fd是不是WebSocket可以用这个方法。

函数原型

```
/**
 * 判断这个fd是不是一个WebSocket连接，用于区分tcp和websocket
 * 握手后才识别为websocket
 * @param $fd
 * @return bool
 * @throws \Exception
 */
public function isWebSocket($fd)
```

isTaskWorker

判断当前进程是不是Task进程。

函数原型：

```
    /**
     * 是否是task进程
     * @return bool
     */
    public function isTaskWorker()
```

getSocketName

获取服务器ip:port。

initAsynPools

1.7.6版本后新增方法

见AppServer中的initAsynPools

addAsynPool

1.7.6版本后新增方法

函数原型：

```
function addAsynPool($name, AsynPool $pool)
```

用于添加一个一步连接池只能用于initAsynPools函数中。

getAsynPool

1.7.6版本后新增方法

函数原型：

```
function getAsynPool($name)
```

用于返回一个异步连接池

一般在controller中或者model中可以通过下面这种方法获取异步连接池

```
get_instance()->getAsynPool('redis2');
```

Loader

加载器

在Controller，Model，Task中经常用到。

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();
    $this->destroy();
}
```

注意

在1.7.5版本之前loader->model不允许出现循环loader，比如A加载了B，B又加载了A，又或者A加载了A，这种会引发严重的服务器故障，出现死循环问题。

1.7.5版本后框架会自动发现这种情况并会将上层的实例直接传回。

比如A->B->C->A这种，C里面的A其实就是最初的A。

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');
        $testTask->test();
        $testTask->startTask(null);
    }
```

需要注意$this->loader->task('TestTask')返回的并不是TestTask的实例，其实返回的是一个TaskProxy，这个TaskProxy是个单例模式，所以调用task的时候不要对TaskProxy缓存，随时用随时获取。

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。

TaskProxy和Task的更加深入的用法见具体章节。

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 $template
     * @return \League\Plates\Template\Template
     */
    public function view($template)
```

例子：

```
    /**
     * html测试
     */
    public function http_html_test()
    {
        $template = $this->loader->view('server::error_404');
        $this->http_output->end($template->render(['controller'=>
'TestController\html_test','message'=>'页面不存在！']));
    }
```

> server::error_404 是指Server/Views下的error_404.php文件
> app::error_404 是指app/Views下的error_404.php文件

具体的请参考Plates模板引擎。

Controller

控制器，重要的组件之一。

__construct

构造函数。

Controller类是对象池模式的，它具有对象回收的机制。

所以在构造函数中初始化的所有对象，如果不是可以复用的请在destory函数中清理，同样声明的类对象也需要在destory函数中进行清理，以免影响回收后的复用。

```
/**
     * 销毁
     */
    public function destroy()
    {
        parent::destroy();
        unset($this->fd);
        unset($this->uid);
        unset($this->client_data);
        unset($this->request);
        unset($this->response);
        $this->http_input->reset();
        $this->http_output->reset();
        ControllerFactory::getInstance()->revertController($this)
;
    }
```

注意

在1.7.5版本后__construct方法底层修改为了final，不再允许继承修改，用户需要将代码移到initialization处。

initialization

初始化方法，每次框架调用控制器方法之前都会执行initialization。

比如TestController/test，当服务器调用test方法之前会先调用initialization方法并传入参数。

$controller_name 准备执行的controller名称

$method_name 准备执行的method名称

```
    /**
     * 初始化每次执行方法之前都会执行initialization
     * @param $controller_name 准备执行的controller名称
     * @param $method_name 准备执行的method名称
     */
    public function initialization($controller_name, $method_name
)
    {

    }
```

注意

在initialization中初始化的非局部变量可以不在destroy中进行销毁操作，因为每次访问都会执行initialization，值会被覆盖。

onExceptionHandle

异常的回调

如果在控制器方法执行过程中抛出了异常，那么会直接跳转到onExceptionHandle方法。

默认我们提供了一个异常处理。

```
    /**
     * 异常的回调
     * @param \Exception $e
     */
    public function onExceptionHandle(\Exception $e)
    {
        switch ($this->request_type) {
            case SwooleMarco::HTTP_REQUEST:
                $this->http_output->end($e->getMessage());
                break;
            case SwooleMarco::TCP_REQUEST:
                $this->send($e->getMessage());
                break;
        }
    }
```

如果需要修改请继承Controller。

实际上我们建议在你开始编写代码之前将Controller，Model, 等常用的类都继承一遍，变成自己业务的基类。这样才能保证框架更新的时候不会出现代码冲突问题。

由于协程的存在，异常嵌套处理比较麻烦，下面将其情况一一说明。
1.普通的controller异常测试，onExceptionHandle将接收到回调

```
/**
     * 普通的controller异常测试
     * @throws \Exception
     */
    public function http_testExceptionHandleII()
    {
        throw new \Exception('ExceptionHandle');
    }
```

2.普通model内的异常测试,onExceptionHandle将接收到回调

```
/**
     * 普通model的异常测试
     * @throws \Exception
     */
    public function http_testExceptionHandleIII()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $result = $this->testModel->test_exception();
    }
```

3.协程的model内的异常测试,onExceptionHandle将接收到回调

```
/**
     * 协程的model报错需要增加try catch捕获，否则无法抛出
     */
    public function http_testExceptionHandleIV()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $result = yield $this->testModel->test_exceptionII();
    }
```

4.总结：无论协程或者是多层协程嵌套，还是普通的代码，或者普通代码混合协程，都可以一路向上throw。并且可以通过try catch捕获。和正常代码没有任何区别。

destory

销毁

在这里我们将在控制器回收之前进行数据清理。

请务必注意，控制器的每一个可能被用户调用的方法都需要执行destory操作，不如不进行destory，那么这个控制器将一直无法被回收，访问量大了之后无法被回收的控制器将越来越多，会导致内存泄露和运行缓慢的严重问题。

好在我们在绝大多数可能会终止Controller生存周期的方法上都默认加入的destory，如果你需要手动控制，请注意所有send/end方法的默认参数。

一旦Controller执行了destory，将不允许进行send/end的操作。所以destory是你逻辑最后执行的一步。

send

用于TCP/WebSocket请求

向当前客户端发送消息

调用的是get_instance的send方法，为数据包加入协议头，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * 向当前客户端发送消息
     * @param $data
     * @param $destory
     * @throws SwooleException
     */
    protected function send($data, $destory = true)
```

sendToUid

用于TCP/WebSocket请求

向对应uid的客户端发送消息

调用的是get_instance的sendToUid方法，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * sendToUid
     * @param $uid
     * @param $data
     * @throws SwooleException
     */
    protected function sendToUid($uid, $data, $destory = true)
```

sendToUids

用于TCP/WebSocket请求

向对应多个uid的客户端发送消息

调用的是get_instance的sendToUids方法，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * sendToUids
     * @param $uids
     * @param $data
     * @param $destory
     * @throws SwooleException
     */
    protected function sendToUids($uids, $data, $destory = true)
```

sendToAll

用于TCP/WebSocket请求

向所有客户端广播消息

调用的是get_instance的sendToAll方法，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * sendToAll
     * @param $data
     * @param $destory
     * @throws SwooleException
     */
    protected function sendToAll($data, $destory = true)
```

sendToGroup

用于TCP/WebSocket请求

向组内客户端广播消息

调用的是get_instance的sendToGroup方法，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * 发送给群
     * @param $groupId
     * @param $data
     * @param bool $destory
     * @throws SwooleException
     */
     protected function sendToGroup($groupId, $data, $destory = true)
```

kickUid

用于TCP/WebSocket请求

剔除Uid对应的客户端

调用的是get_instance的kickUid方法，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * 踢用户
     * @param $uid
     */
    protected function kickUid($uid)
```

bindUid

用于TCP/WebSocket请求

绑定Uid到对应的客户端

调用的是get_instance的bindUid方法，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * bindUid
     * @param $fd
     * @param $uid
     * @param bool $isKick
     */
    protected function bindUid($fd, $uid, $isKick = true)
```

例子

```
$this->bindUid($this->fd,10000);
```

unBindUid

用于TCP/WebSocket请求

解除绑定Uid到对应的客户端

调用的是get_instance的unBindUid方法，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * unBindUid
     * @param $uid
     */
    protected function unBindUid($uid)
```

close

用于TCP/WebSocket请求

断开当前fd的连接

调用的是get_instance的close方法，但会默认自动销毁，并且在单元测试中会记录用户的操作。

函数原型：

```
    /**
     * 断开链接
     * @param $fd
     * @param bool $autoDestory
     */
    protected function close($fd, $autoDestory = true)
```

$redis_pool

获取redis连接池对象

例子：

```
    /**
     * Redis decrBy 命令
     */
    public function testRedisDecrBy()
    {
        yield $this->redis_pool->getCoroutine()->set('key1', 10);

        yield $this->redis_pool->getCoroutine()->decrBy('key1',
10);
        $value = yield $this->redis_pool->getCoroutine()->get('k
ey1');
        $this->destory();
    }
```

$mysql_pool

获取mysql连接池实例

例子：

```
    /**
     * mysql Replace 命令
     */
    public function testMysqlReplace()
    {
        $value = yield $this->mysql_pool->dbQueryBuilder->replace
('MysqlTest')
            ->set('firstname', 'White')
            ->set('lastname', 'Cat')
            ->set('age', '26')
            ->set('townid', '10000')->coroutineSend();
        $this->destory();
    }
```

$http_input

适用于Http请求。

对应HttpInput类。

$http_output

适用于Http请求。

对应HttpOutPut类。

$request_type

请求的类型是TCP还是HTTP。

具有俩个常量

```
SwooleMarco::TCP_REQUEST;
SwooleMarco::HTTP_REQUEST;
```

这里需要针对WebSocket说明下，由于WebSocket同样是长连接，所以在框架中WebSocket和TCP所使用的方法是一致的，开发者不需要关心这个连接是WebSocket还是TCP，框架会自行处理。

$fd

获取当前连接的fd。

uid

如果当前连接有绑定uid，这里将返回uid。

正确的uid是不为0的整数。

$client_data

用于TCP/WebSocket

获取用户请求的协议体。

$request

这里获取的是swoole的swoole_http_request。

一般情况下不需要用到，在HttpInput中已经封装好了。

$response

这里获取的是swoole的swoole_http_response。

一般情况下不需要用到，在HttpOutput中已经封装好了。

$loader

获取Loader的一个实例。

$logger

获取一个Logger的实例。

这个logger是Monolog/Logger。

$server

获取的是swoole的swoole_server。

$config

获取的是Config的实例，这里的Config对应config文件夹中所有的配置。

config是 Noodlehaus/Config。

$client

HttpClient，发送http请求。

例子：

```
    /**
     * 协程的httpclient测试
     */
    public function http_test_httpClient()
    {
        $httpClient = yield $this->client->coroutineGetHttpClient
('http://localhost:8081');
        $result = yield $httpClient->coroutineGet("/TestControll
er/test_request", ['id' => 123]);
        $this->http_output->end($result);
    }
```

defaultMethod

当控制器方法不存在的时候的默认方法

1.7.3版本对NormalRoute做了些调整，首先先匹配Controller，如果找不到直接返回404，然后寻找Method，如果找不到则会执行defaultMethod方法。

我们默认提供了一个处理机制。

```
    /**
     * 当控制器方法不存在的时候的默认方法
     */
    public function defaultMethod()
    {
        if ($this->request_type == SwooleMarco::HTTP_REQUEST) {
            $this->http_output->set_header('HTTP/1.1', '404 Not
Found');
            $template = $this->loader->view('server::error_404');

            $this->http_output->end($template->render());
        } else {
            throw new SwooleException('method not exist');
        }
    }
```

如果需要自定义请重写此方法。

getContext

1.7.8新增的功能

用于获取上下文。

在initialization时会为Controller创建一个Context数组对象，并自动为添加了request_id的字段，用于区别其他的请求。

函数原型：

```
/**
 * 获取上下文
 * @return array
 */
public function &getContext()
{
    return $this->context;
}
```

通过这个方法获取到的是context的引用，对其的修改会影响到拥有这个引用的值。

通过loader->model,loader->task，会将Context传递下去，Model持有Context的引用，而Task持有Context的副本。

在Model和Task中可以通过getContext获取到上下文。

Model

Model是专门和数据打交道的模块。

协程模式

model中可以调用redis，mysql，task的协程模式，但是请注意，如果model是使用协程的，那么controller或者model调用这个包含协程model接口时，也要加上yeild关键字。
例子：

```
public function http_testCoroutine()
    {
        $this->testModel = $this->loader->model('TestModel', $th
is);
        $result = yield $this->testModel->test_coroutine();
        $this->http_output->end($result);
    }
```

Model层级

1.7.4版本支持深层级的Model。

```
$this->testModel = $this->loader->model('Test/TestModel', $this);
```

如上会调用Test目录下TestModel，请注意命名空间的规范。

在1.7.4版本之前需要这样

```
$this->testModel = $this->loader->model('Test\\TestModel', $this)
;
```

__construct

构造函数。

Model类是对象池模式的，它具有对象回收的机制。

所以在构造函数中初始化的所有对象，如果不是可以复用的请在destory函数中清理，同样声明的类对象也需要在destory函数中进行清理，以免影响回收后的复用。

```
/**
 * 销毁回归对象池
 */
public function destroy()
{
    parent::destroy();
    ModelFactory::getInstance()->revertModel($this);
}
```

注意

在1.7.5版本后__construct方法底层修改为了final，不再允许继承修改，用户需要将代码移到initialization处。

destory

销毁

在这里我们将在控制器回收之前进行数据清理。

$redis_pool

获取redis连接池对象

例子：

```
    /**
     * Redis decrBy 命令
     */
    public function testRedisDecrBy()
    {
        yield $this->redis_pool->getCoroutine()->set('key1', 10);

        yield $this->redis_pool->getCoroutine()->decrBy('key1',
10);
        $value = yield $this->redis_pool->getCoroutine()->get('k
ey1');
        $this->destory();
    }
```

$mysql_pool

获取mysql连接池实例

例子：

```
    /**
     * mysql Replace 命令
     */
    public function testMysqlReplace()
    {
        $value = yield $this->mysql_pool->dbQueryBuilder->replace
('MysqlTest')
            ->set('firstname', 'White')
            ->set('lastname', 'Cat')
            ->set('age', '26')
            ->set('townid', '10000')->coroutineSend();
        $this->destory();
    }
```

$client

HttpClient，发送http请求。

例子：

```
    /**
     * 协程的httpclient测试
     */
    public function http_test_httpClient()
    {
        $httpClient = yield $this->client->coroutineGetHttpClient
('http://localhost:8081');
        $result = yield $httpClient->coroutineGet("/TestControll
er/test_request", ['id' => 123]);
        $this->http_output->end($result);
    }
```

initialization

1.7.5版本新增的方法，用于代替__construct

初始化方法，每次框架loader->model时都会执行initialization。

```
    /**
     * 当被loader时会调用这个方法进行初始化
     */
    public function initialization()
    {

    }
```

注意

在initialization中初始化的非局部变量可以不在destroy中进行销毁操作，因为每次loader->model都会执行initialization，值会被覆盖。

虽然可以在在initialization时一次性loader所有的model并进行成员变量的缓存，但是如果追求更高的效率,loader的方法应该在使用这个model之前获取。

View

视图，适用于http

模板引擎参考plates

Task

Task是异步任务模块服务于controller和model。

1.7以后的版本，Task中可以使用协程，所以写法和Model基本一致。

Task类在task进程中是单例，每种类只会实例化一次。

TaskProxy

> controller和model中获取的Task其实是TaskProxy

- startTask

```
/**
* 开始异步任务
*/
function startTask($callback,$id = -1)
```

- startTaskWait

```
/**
* 开始同步任务
*/
function startTaskWait($timeOurt = 0.5, $id = -1)
```

Controller和Model中的用法

AppTask.php

```
    class AppTask extends Task
    {
        public function testTask()
        {
            return "test task\n";
        }

    }
```

AppController.php

```
    class AppController extends Controller
    {
        public function http_test_task()
        {
            $AppTask = $this->loader->task('AppTask');
            $AppTask->testTask();
            $AppTask->startTask(function ($serv, $task_id, $data)
{
                $this->http_output->end($data);
            });
        }
    }
```

TimerTask的用法

添加到配置文件timerTask.php中

```
$config['timerTask'][] = [
    'task_name'=>'AppTask',
    'method_name'=>'testTask',
    'start_time'=>'Y-m-d 00:00:00',
    'end_time'=>'Y-m-d 23:59:59',
    'interval_time'=>'2',
];
```

协程模式

示例：

```
        $task = $this->loader->task('TestTask');
        $task->test();
        $result = yield $task->coroutineSend();
```

使用协程会大大简化代码的书写，提高代码的可读性。

上面的代码通过yield关键字返回了异步回调的值。

如果你的***task***是没有结果返回的，请不要用协程，这样会超时。

Task层级

1.7.4版本支持深层级的Task。

```
$this->testTask = $this->loader->task('Test/TestTask');
```

如上会调用Test目录下TestTask，请注意命名空间的规范。

在1.7.4版本之前需要这样

```
$this->testTask = $this->loader->task('Test\\TestTask');
```

耗时Task的中断

1.7.7版本后具有此功能

步骤如下：

1.获取task_id

```
public function http_startInterruptedTask()
{
    $testTask = $this->loader->task('TestTask');
    //这里testInterrupted是TestTask的一个方法
    $task_id = $testTask->testInterrupted();
    $testTask->startTask(null);
    $this->http_output->end("task_id = $task_id");
}
```

在调用Task的方法时会返回一个task_id。

也可以通过

```
get_instance()->getServerAllTaskMessage();
```

获取到所有正在运行的Task信息。

2.Task需要设置中断检查

```
//TestTask.php文件
/**
 * 测试中断
 */
public function testInterrupted()
{
    while (true) {
        if ($this->checkInterrupted()) {
            print_r("task已中断\n");
            break;
        }
    }
}
```

这里的testInterrupted为上面调用的方法，在这个方法循环中加入

```
$this->checkInterrupted()
```

如果返回为true代表触发了中断，你需要结束循环。

3.中断Task

```
public function http_interruptedTask()
{
    $task_id = $this->http_input->getPost('task_id');
    get_instance()->interruptedTask($task_id);
    $this->http_output->end("ok");
}
```

调用

```
get_instance()->interruptedTask($task_id);
```

来中断一个正在运行的task。

HttpInput

注意1.7.4版本有些方法名修改了，统一了驼峰。

postGet

优先先从post中取，没有再从get中取

xss_clean:是否使用xss清理

函数原型：

```
/**
 * postGet
 * @param $index
 * @param $xss_clean
 * @return string
 */
public function postGet($index, $xss_clean = true)
```

post

从post中取

xss_clean:是否使用xss清理

函数原型：

```
/**
 * post
 * @param $index
 * @param $xss_clean
 * @return string
 */
public function post($index, $xss_clean = true)
```

get

从get中取

xss_clean:是否使用xss清理

函数原型：

```
    /**
     * get
     * @param $index
     * @param $xss_clean
     * @return string
     */
    public function get($index, $xss_clean = true)
```

getPost

优先先从get中取，没有再从post中取

xss_clean:是否使用xss清理

函数原型：

```
/**
 * getPost
 * @param $index
 * @param $xss_clean
 * @return string
 */
public function getPost($index, $xss_clean = true)
```

getAllPostGet

获取所有的Post和Get，返回一个数组

函数原型：

```
    /**
     * 获取所有的post和get
     */
    public function getAllPostGet()
```

getAllHeader

获取所有的Header

getRawContent

获取原始的POST包体

cookie

获取cookie数据

函数原型：

```
    /**
     * cookie
     * @param $index
     * @param $xss_clean
     * @return string
     */
    public function cookie($index, $xss_clean = true)
```

getRequestHeader

获取头信息

函数原型：

```
    /**
     * getRequestHeader
     * @param $index
     * @param $xss_clean
     * @return string
     */
    public function getRequestHeader($index, $xss_clean = true)
```

server

获取server信息

函数原型:

```
    /**
     * 获取Server相关的数据
     * @param $index
     * @param bool $xss_clean
     * @return array|bool|string
     */
    public function server($index, $xss_clean = true)
```

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

GET/POST。。

getRequestUri

获取请求的Uri

getPathInfo

获取请求的Path

HttpOutput

注意1.7.4版本有些方法名修改了，统一了驼峰。

setStatusHeader

设置返回的状态码

函数原型：

```
    /**
     * Set HTTP Status Header
     *
     * @param    int    the status code
     * @param    string
     * @return HttpOutPut
     */
    public function setStatusHeader($code = 200)
```

setContentType

设置返回的Content-Type

函数原型

```
    /**
     * Set Content-Type Header
     *
     * @param    string $mime_type Extension of the file we're o
utputting
     * @return    HttpOutPut
     */
    public function setContentType($mime_type)
```

setHeader

设置返回的头信息

函数原型

```
    /**
     * Set Content-Type Header
     *
     * @param    string $mime_type Extension of the file we're o
utputting
     * @return    HttpOutPut
     */
    public function setContentType($mime_type)
```

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

函数原型：

```
    /**
     * 发送
     * @param string $output
     * @param bool $gzip
     * @param bool $destory
     */
    public function end($output = '', $gzip = true, $destory = t
rue)
```

setCookie

设置cookie。

函数原型：

```
    /**
     * 设置HTTP响应的cookie信息。此方法参数与PHP的setcookie完全一致。
     * @param string $key
     * @param string $value
     * @param int $expire
     * @param string $path
     * @param string $domain
     * @param bool $secure
     * @param bool $httponly
     */
    public function setCookie(string $key, string $value = '', i
nt $expire = 0, string $path = '/', string $domain = '', bool $s
ecure = false, bool $httponly = false)
```

endFile

发送文件

函数原型：

```
    /**
     * 输出文件
     * @param $root_file
     * @param $file_name
     * @param bool $destory
     * @return mixed
     */
    public function endFile($root_file, $file_name,$destory = tr
ue)
```

RedisAsynPool

异步redis连接池

示例：

```
$this->redis_pool->hMGet('test',[1,2,3,4], function ($uids) {
});
```

协程模式

```
        $this->redis_pool->getCoroutine();
```

这个函数在注释中表示返回的是\Redis扩展的一个抽象类其实返回的是一个迭代器，我们只需要他的代码提示，1.7版本已经将同步和异步的使用方式做了完全的统一，和\Redis扩展的用法完全一致。

示例：

```
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('uid', 10303)->coroutineSend();
        $mysql_result = yield $mySqlCoroutine;
        $redisCoroutine = $this->redis_pool->getCoroutine()->get(
'test');
        $redis_result = yield $redisCoroutine;
```

使用协程会大大简化代码的书写，提高代码的可读性。 上面的代码通过yield关键字返回了异步回调的值。 执行顺序 mysql_send->mysql_rev->redis_send->redis_rev;

```
        $mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->sel
ect('*')->from('account')->where('uid', 10303)->coroutineSend();

        $redisCoroutine = $this->redis_pool->getCoroutine()->get(
'test');
        $mysql_result = yield $mySqlCoroutine;
        $redis_result = yield $redisCoroutine;
```

执行顺序 mysql_send->redis_send->mysql_rev->redis_rev;

MysqlAsynPool

异步mysql连接池

普通用法

示例：

```
/**
     * mysql 测试
     * @throws \Server\CoreBase\SwooleException
     */
    public function mysql_test()
    {
        $this->mysql_pool->dbQueryBuilder->select('*')->from('ac
count')->where('sex', 1);
        $this->mysql_pool->query(function ($result) {
            print_r($result);
        });
        $this->destroy();
    }
```

其中dbQueryBuilder对应的是Miner，关于用法参考Miner

协程事务

非协程版事务相对复杂不建议使用。

```
/**
     * mysql 事务协程测试
     */
    public function http_mysql_begin_coroutine_test()
    {
        $id = yield $this->mysql_pool->coroutineBegin($this);
        $update_result = yield $this->mysql_pool->dbQueryBuilder->update('user_info')->set('sex', '0')->where('uid', 36)->coroutineSend($id);
        $result = yield $this->mysql_pool->dbQueryBuilder->select('*')->from('user_info')->where('uid', 36)->coroutineSend($id);
        if ($result['result'][0]['channel'] == 888) {
            $this->http_output->end('commit');
            yield $this->mysql_pool->coroutineCommit($id);
        } else {
            $this->http_output->end('rollback');
            yield $this->mysql_pool->coroutineRollback($id);
        }
    }
```

协程模式

示例：

```
$mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select('*')->from('account')->where('uid', 10303)->coroutineSend();
        $mysql_result = yield $mySqlCoroutine;
        $redisCoroutine = $this->redis_pool->coroutineSend('get', 'test');
        $redis_result = yield $redisCoroutine;
```

使用协程会大大简化代码的书写，提高代码的可读性。
上面的代码通过yield关键字返回了异步回调的值。
执行顺序 mysql_send->mysql_rev->redis_send->redis_rev;

```
$mySqlCoroutine = $this->mysql_pool->dbQueryBuilder->select('*')-
>from('account')->where('uid', 10303)->coroutineSend();
        $redisCoroutine = $this->redis_pool->coroutineSend('get',
 'test');
        $mysql_result = yield $mySqlCoroutine;
        $redis_result = yield $redisCoroutine;
```

执行顺序 mysql_send->redis_send->mysql_rev->redis_rev;

获取mysql语句

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

关于WebSocket的一些说明

在SD框架中，WebSocket，HTTP，TCP这3中协议都可以同时开启。

须知

1.HTTP和WebSocket监听的是同一个端口，本身WebSocket就是Http协议的一中，HTTP协议通过握手转变成了WebSocket。可以在SD启动界面中看到WebSocket和HTTP端口是一致的

2.WebSocket和TCP一样是属于长连接，1个用户和SD服务器保持连接是通过fd来识别的，和TCP一样WebSocket区分客户端连接也是通过fd，那么binduid，unbinduid函数也适用于WebSocket，并且为了方便SD的Send族函数是同时适配TCP和WebSocket，通过Send族函数向一个fd发送信息时SD会自动判断连接是TCP还是WebSocket。

3.WebSocket和TCP的封装器保持一致，在函数名的体现上也保持一致。

总结

在SD框架中不需要刻意区分TCP和WebSocket，可以简单的理解为WebSocket的所有特性和方法都和TCP保持一致，他们公用一套API，他们公用一套封装器。

分布式

SD框架使用分布式非常简单。

短连接

短连接指的是HTTP/HTTPS连接，这类连接可以通过Nginx或者LVS进行集群化，通过Redis来共享用户信息，这里不再多说。

长连接

SD的长连接分布式使用起来也非常简单，只需满足一个条件

*保证所有的服务器都存在于一个内网环境并且处于同一网段。

在阿里云上可以使用VPN虚拟内网保证所有服务器都在一个网段。

然后在其中一台主机上启动

```
php start_swoole_dispatch.php start -d
```

dispatch和所有的服务器会自动连接。

通过LVS+KeepAlived进行负载均衡

LVS搭建四层负载均衡，暴露一个VIP，客户端通过连接VIP实现负载均衡

2-10台机器的集群模式

配置好LVS和keeplived用于服务器组的负载均衡，dispatch服务器和从redis安装到同一个物理机上之间使用unixsock进行通讯，server服务单独部署在一台物理机上，主redis单独部署在一台物理机上，一般5台以下的server只需要搭配一个dispatch，5台以上可以搭配2个dispatch，2个dispatch服务器才有必要做redis的主从。

注：dispatch服务器只会读redis完全不会写入redis。

10台以上的集群模式

这种可能性能的瓶颈主要堆积到redis的读上了，主从读写分离这种模式只能一定成程度上提高效率，出现redis瓶颈就需要进行redis集群的搭建了。

建议dispatch服务器要比server服务先启动，否则server寻找dispatch服务器会有30秒的延迟。

客户端实现负载均衡

客户端实现算法（随机/一致性hash。。）通过算法得到一个服务器ip进行连接，实现负载均衡

LVS+Keepalive 比较详细的安装配置文档

LVS说明:

目前有三种IP负载均衡技术（VS/NAT、VS/TUN和VS/DR,八种调度算法（rr,wrr,lc,wlc,lblc,lblcr,dh,sh）。

在 调度器的实现技术中，IP负载均衡技术是效率最高的。在已有的IP负载均衡技术中有通过网络地址转换（Network Address Translation）将一组服务器构成一个高性能的、高可用的虚拟服务器，我们称之为VS/NAT技术（Virtual Server via Network Address Translation），大多数商品化的IP负载均衡调度器产品都是使用此方法，如Cisco的LocalDirector、F5的Big/IP和 Alteon的ACEDirector。在分析VS/NAT的缺点和网络服务的非对称性的基础上，我们提出通过IP隧道实现虚拟服务器的方法VS/TUN （Virtual Server via IP Tunneling），和通过直接路由实现虚拟服务器的方法VS/DR（Virtual Server via Direct Routing），它们可以极大地提高系统的伸缩性。所以，IPVS软件实现了这三种IP负载均衡技术，它们的大致原理如下（我们将在其他章节对其工作原 理进行详细描述）

Virtual Server via Network Address Translation（VS/NAT）

通过网络地址转换，调度器重写请求报文的目标地址，根据预设的调度算法，将请求分派给后端的眞实服务器；眞实服务器的响应报文通过调度器时，报文的源地址被重写，再返回给客户，完成整个负载调度过程。

Virtual Server via IP Tunneling（VS/TUN）

采用NAT技术时，由于请求和响应报文都必须经过调 度器地址重写，当客户请求越来越多时，调度器的处理能力将成为瓶颈。为了解决这个问题，调度器把请求报文通过IP隧道转发至眞实服务器，而眞实服务器将响 应直接返回给客户，所以调度器只处理请求报文。由于一般网络服务应答比请求报文大许多，采用 VS/TUN技术后，集群系统的最大吞吐量可以提高10倍。

Virtual Server via Direct Routing（VS/DR）

VS/DR通过改写请求报文的MAC地址，将请求 发送到眞实服务器，而眞实服务器将响应直接返回给客户。同VS/TUN技术一样，VS/DR技术可极大地提高集群系统的伸缩性。这种方法没有IP隧道的开 销，对集群中的眞实服务器也没有必须支持IP隧道协议的要求，但是要求调度器与眞实服务器都有一块网卡连在同一物理网段上。

针对不同的网络服务需求和服务器配置，IPVS调度器实现了如下八种负载调度算法：使用比较多的是以下四种：

轮叫（Round Robin）

调度器通过"轮叫"调度算法将外部请求按顺序轮流分配到集群中的眞实服务器上，它均等地对待每一台服务器，而不管服务器上实际的连接数和系统负载。

加权轮叫（Weighted Round Robin）

调度器通过"加权轮叫"调度算法根据眞实服务器的不同处理能力来调度访问请求。这样可以保证处理能力强的服务器处理更多的访问流量。调度器可以自动问询眞实服务器的负载情况，并动态地调整其权值。

最少链接（Least Connections）

调度器通过"最少连接"调度算法动态地将网络请求调度到已建立的链接数最少的服务器上。如果集群系统的眞实服务器具有相近的系统性能，采用"最小连接"调度算法可以较好地均衡负载。

加权最少链接（Weighted Least Connections）

在集群系统中的服务器性能差异较大的情况下，调度器采用"加权最少链接"调度算法优化负载均衡性能，具有较高权值的服务器将承受较大比例的活动连接负载。调度器可以自动问询眞实服务器的负载情况，并动态地调整其权值。

1、拓扑描述：(一定要理解这个拓扑关系） 负载服务器master眞实IP 192.168.1.252 负载服务器backup眞实IP 192.168.1.230 负载服务器虚拟IP 192.168.1.229 后端WEB服务器IP 192.168.1.220 后端WEB服务器IP 192.168.1.231 2、升级内核

```
#yum install kernel
```

3、重启服务器，使用新的内核 4、删除旧版、升级新版内核

```
#rpm -e kernel-2.6.18-53.el5
#rpm -e kernel-devel-2.6.18-53.el5
#rpm -e kernel-headers-2.6.18-53.el5 --nodeps
#yum install kernel-headers
#yum install kernel-devel
```

5、下载软件

```
#wget http://www.linuxvirtualser... ... ipvsadm-1.24.tar.gz
#wget http://www.keepalived.org/...
```

6、安装ipvsadm-1.24 //master和backup

```
#        rpm -ivh ipvsadm-1.24-6.src.rpm
#        cd /usr/src/redhat/SOURCES
#        tar -zxvf ipvsadm-1.24.tar.gz
#        cd ipvsadm
#       uname -r  //查询版本
       2.6.18-53.el5xen
#        ln -s  /usr/src/kernels/2.6.18-53.el5xen-i686/  /usr/sr
c/linux  //假如这里的内核版本不一样的话，make的时候会出现错误。
#        make;make install
```

7,安装keepalived. 在负载均衡服务器上执行 master和backup 1、解压

```
        #tar -zxvf keepalived-1.1.15.tar.gz
        #cd keepalived-1.1.15
        #./configure --prefix=/usr/local/keepalived
        #make;make install
        #cp /usr/local/keepalived/sbin/rc.d/init.d/keepalived /sb
in/rc.d/init.d/
        #cp /usr/local/keepalived/sbin/sysconfig/keepalived /sbin
/sysconfig/
        #mkdir /sbin/keepalived
        #cp /usr/local/keepalived/sbin/keepalived/keepalived.conf
 /sbin/keepalived/
        #cp /usr/local/keepalived/sbin/keepalived /usr/sbin/
        #service keepalived start|stop
```

8、开启负载服务器路由机制 //master和backup

```
#         vi /sbin/sysctl.conf 保证有如下内容
        net.ipv4.ip_forward = 1
        执行
#         sysctl -p
```

9、建立负载服务器启动脚本 //master和backup

```
#vi /sbin/lvsdr.sh
#!/bin/bash
VIP=192.168.1.229
RIP1=192.168.1.220
RIP2=192.168.1.231
/sbin/rc.d/init.d/functions
case "$1" in
start)
       echo "start LVS of DirectorServer"
       #Set the Virtual IP Address
       /sbin/ifconfig eth0:1 $VIP broadcast $VIP netmask 255.255
.255.255 up
       /sbin/route add -host $VIP dev eth0:1
       #Clear IPVS Table
       /sbin/ipvsadm -C
       #Set Lvs
       /sbin/ipvsadm -A -t $VIP:80 -s wrr
       /sbin/ipvsadm -a -t $VIP:80 -r $RIP1:80 -g
       /sbin/ipvsadm -a -t $VIP:80 -r $RIP2:80 -g
       #Run Lvs
       /sbin/ipvsadm
;;
stop)
echo "Close LVS Directorserver"
/sbin/ifconfig eth0:1 down
/sbin/ipvsadm -C
;;
*)
echo "Usage0{start|stop}"
exit 1
esac
```

10、分配权限

```
#chmod 755 /sbin/lvsdr.sh
```

11、执行测试

```
#         /sbin/lvsdr.sh start
        查看ifconfig是否有ifcfg-eth0:1   (有就对了)
        查看route -n 路由表是否多了eth0:1路由(有就对了)
#         /sbin/lvsdr.sh stop
        查看ifconfig是否有ifcfg-eth0:1   (无就对了)
        查看route -n 路由表是否多了eth0:1路由(无就对了)
#         /sbin/lvsdr.sh  adsa
        是否提示参数错误,只能使用{start|stop}。
```

12、配置后端WEB服务器

```
        在192.168.1.231和192.168.1.220上分别建立如下脚本。
#         vi /sbin/realdr.sh
        #!/bin/bash
        VIP=192.168.1.229
        /sbin/ifconfig lo:0 $VIP broadcast $VIP netmask 255.255.2
55.255 up
        /sbin/route add -host $VIP dev lo:0
        echo "1">/proc/sys/net/ipv4/conf/default/arp_ignore
        echo "2">/proc/sys/net/ipv4/conf/default/arp_announce
        echo "1">/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "2">/proc/sys/net/ipv4/conf/all/arp_announce
        sysctl -p
```

13、配置权限

```
#         chmod 755 /sbin/realdr.sh
```

14、在两台web服务器上分别执行其指命。

```
/sbin/realdr.sh start
```

15.配置keepalived.conf配置文件 //master和backup

```
#vi /etc/keepalived/keepalived.conf
! Configuration File for keepalived
global_defs {
```

```
  notification_email {
    dalianlxw@139.com
  }
  notification_email_from xwluan@tsong.cn
  smtp_server 222.73.214.147
  smtp_connect_timeout 30
  router_id LVS_DEVEL
}
vrrp_instance VI_1 {
   state MASTER  //备份服务器设置为backup
   interface eth0
   virtual_router_id 51
   priority 100  ////备份服务器设置小于100
   advert_int 1
   authentication {
       auth_type PASS
       auth_pass 1111
   }
   virtual_ipaddress {
       192.168.1.229
   }
}
virtual_server 192.168.1.229 80 {
   delay_loop 6  //隔6秒查询
   lb_algo wrr   //lvs算法
   lb_kind DR    //(Direct Route)
   persistence_timeout 60  //同一IP的连接60秒内被分配到同一台realserv
er
   inhibit_on_failure  //当web挂掉的时候，前面请求的用户，可以继续打开
网页，但是后面的请求不会调度到挂掉的web上面。
   protocol TCP   //用TCP协议检查realserver状态
   real_server 192.168.1.220 80 {
       weight 3    //权重
       TCP_CHECK {
           connect_timeout 10   //10秒无响应超时
           nb_get_retry 3
           delay_before_retry 3
       }
   }
   real_server 192.168.1.231 80 {
       weight 1
       TCP_CHECK {
```

```
            connect_timeout 10
            nb_get_retry 3
            delay_before_retry 3
        }
    }

}
```

16,启动keepalived

```
# /etc/rc.d/init.d/keepalived start
```

17:设置成自启动

```
#vi /etc/rc.local //里面添加
/etc/init.d/keepalived restart
/etc/lvsdr.sh start
```

测试算法：

我的测试环境中，算法使用的是wrr,和wlc这两种。

权重问题：

当lvs配置文件lvs-dr.sh改变权重以及keepalived配置文件keepalived.conf修改权重后，哪个文件重新启动，哪个文件的权重生效。同时权重在master和backup上面可以设置不同。

当 算法是rr的时候，权重没有作用，但是当算法是wlc和wrr的时候，必须设置权重，可以根据服务器的性能和配置，来确定权重的大小，当权重大的时 候，lvs调度的服务也就多，同时权重高的服务器先收到链接。当小的时候，lvs调度的比较少。当权重为0的时候，表示服务器不可用，

测试LVS

1 当我把master的lvs服务关掉的时候，会将用户请求自动切换到backup上面进行工作。

2 当我把web服务关掉的时候，lvs上面会显示web消失，当启用后，web会自动显示 web关闭后：

```
[root@localhost ~]# ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddressort Scheduler Flags
 -> RemoteAddressort           Forward Weight ActiveConn InActCo
nn
TCP  192.168.1.229:80 wlc
 -> 192.168.1.231:80             Route   10     0          0

[root@localhost ~]# ipvsadm -ln
```

web启用后：

```
root@localhost ~]# ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddressort Scheduler Flags
 -> RemoteAddress:Port           Forward Weight ActiveConn InAct
Conn
TCP  192.168.1.229:80 wlc
 -> 192.168.1.231:80             Route   1      0          0

 -> 192.168.1.220:80             Route   1      0          0
```

3 当master服务器down的时候，backup自动会接替服务，当master起来的时候，backup会自动断掉。

常见问题

如何使用HTTPS

配置文件config.php

```
$config['server']['set'] = [
    '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,
    '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
];
```

server.set这个参数其实是Swoole的服务器配置参数，通过在这里面增加https的配置即可实现https。

ssl_cert_file

设置SSL隧道加密，设置值为一个文件名字符串，制定cert证书和key的路径。

https应用浏览器必须信任证书才能浏览网页

wss应用中，发起WebSocket连接的页面必须使用https

浏览器不信任SSL证书将无法使用wss

使用SSL必须在编译swoole时加入--enable-openssl选项

```
$config['server']['set'] = [
    //...
    'ssl_cert_file' => __DIR__.'/config/ssl.crt',
    'ssl_key_file' => __DIR__.'/config//ssl.key',
];
```

ssl_method

设置OpenSSL隧道加密的算法。Server与Client使用的算法必须一致，否则SSL/TLS握手会失败，连接会被切断。

默认算法为 SWOOLE_SSLv23_METHOD

```
$config['server']['set'] = [
    //...
    'ssl_method' => SWOOLE_SSLv3_CLIENT_METHOD,
];
```

如何使用多个**redis**或者多个**mysql**连接

1.7.6版本后SD框架加入了连接池管理功能，我们可以简单的管理多个连接池来实现多个redis连接或者多个mysql连接。

下面分步骤说明。

配置

首先我们在配置中加入多个连接配置如下面展示的redis.php

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

/**
 * 本地环境
 */
$config['redis']['test']['ip'] = '192.168.21.10';
$config['redis']['test']['port'] = 6379;
$config['redis']['test']['select'] = 1;
$config['redis']['test']['password'] = '123456';
$config['redis']['asyn_max_count'] = 10;

/**
 * 本地环境2
 */
$config['redis']['test2']['ip'] = '192.168.21.10';
$config['redis']['test2']['port'] = 6379;
$config['redis']['test2']['select'] = 2;
$config['redis']['test2']['password'] = '123456';
$config['redis']['asyn_max_count'] = 10;

/**
 * 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;
```

其中active所标明的配置代表默认配置，服务器会自动加载使用，通过controller或者其他地方访问到的redisPool就是使用了次配置。

这里我们额外增加了test2，和dispatch配置，其中dispatch配置是给dispatch服务器使用的本地只读redis，这个配置请不要删除。

初始化

接下来我们在AppServer中进行初始化操作

```
/**
 * 这里可以进行额外的异步连接池，比如另一组redis/mysql连接
 * @return array
 */
public function initAsynPools()
{
    parent::initAsynPools();
    $this->addAsynPool('redis2', new RedisAsynPool($this->config,
 'test2'));
}
```

这样我们便加载了名为test2的Redis配置，产生了一个名为redis2的地址池。

使用

我们在Controller中可以使用这个地址池。

```
class AppController extends Controller
{
    /**
     * @var RedisAsynPool
     */
    private $redis2;

    public function initialization($controller_name, $method_name
)
    {
        parent::initialization($controller_name, $method_name);
        $this->redis2 = get_instance()->getAsynPool('redis2');
    }

    /**
     * http测试2个redis
     */
    public function http_test()
    {
        yield $this->redis_pool->getCoroutine()->set('redispool',
 11);
        yield $this->redis2->getCoroutine()->set('redispool', 22)
;
        $result = yield $this->redis_pool->getCoroutine()->get('
redispool');
        print_r($result);
        $result = yield $this->redis2->getCoroutine()->get('redi
spool');
        print_r($result);
        $this->http_output->end($this->AppModel->test());
    }
}
```

通过get_instance()->getAsynPool来获取地址池，然后像往常的方式使用它。