PHP高级编程之消息队列

http://netkiller.github.io/journal/php.mq.html

Mr. Neo Chen (陈景峯), netkiller, BG7NYT

中国广东省深圳市龙华新区民治街道溪山美地
518131
+86 13113668890

<netkiller@msn.com>

2017-06-16

摘要

2015-10-19 第一版

2016-11-31 第二版

目录

1. 什么是消息队列

消息队列（英语：Message queue）是一种进程间通信或同一进程的不同线程间的通信方式

2. 为什么使用消息队列

消息队列技术是分布式应用间交换信息的一种技术。消息队列可驻留在内存或磁盘上,队列存储消息直到它们被应用程序读出。通过消息队列，应用程序可独立地执行，它们不需要知道彼此的位置、或在继续执行前不需要等待接收程序接收此消息。

3. 什么场合使用消息队列

你首先需要弄清楚，消息队列与远程过程调用的区别，在很多读者咨询我的时候，我发现他们需要的是RPC(远程过程调用)，而不是消息队列。

消息队列有同步或异步实现方式，通常我们采用异步方式使用消息队列，远程过程调用多采用同步方式。

MQ与RPC有什么不同？ MQ通常传递无规则协议，这个协议由用户定义并且实现存储转发；而RPC通常是专用协议，调用过程返回结果。

4. 什么时候使用消息队列

同步需求，远程过程调用(PRC)更适合你。

异步需求，消息队列更适合你。

目前很多消息队列软件同时支持RPC功能，很多RPC系统也能异步调用。

消息队列用来实现下列需求

1. 存储转发
2. 分布式事务
3. 发布订阅
4. 基于内容的路由
5. 点对点连接

5. 谁负责处理消息队列

通常的做法，如果小的项目团队可以有一个人实现，包括消息的推送，接收处理。如果大型团队，通常是定义好消息协议，然后各自开发各自的部分， 例如一个团队负责写推送协议部分，另一个团队负责写接收与处理部分。

那么为什么我们不讲消息队列框架化呢？

框架化有几个好处：

1. 开发者不用学习消息队列接口
2. 开发者不需要关心消息推送与接收
3. 开发者通过统一的API推送消息
4. 开发者的重点是实现业务逻辑功能

6. 怎么实现消息队列框架

下面是作者开发的一个SOA框架，该框架提供了三种接口，分别是SOAP,RESTful,AMQP(RabbitMQ)，理解了该框架思想，你很容易进一步扩展，例如增加XML-RPC, ZeroMQ等等支持。

https://github.com/netkiller/SOA

本文只讲消息队列框架部分。

6.1. 守护进程

消息队列框架是本地应用程序（命令行程序），我们为了让他在后台运行，需要实现守护进程。

https://github.com/netkiller/SOA/blob/master/bin/rabbitmq.php

每个实例处理一组队列，实例化需要提供三个参数，$queueName = '队列名', $exchangeName = '交换名', $routeKey = '路由'

```
$daemon = new \framework\RabbitDaemon($queueName = 'email', $exchangeName = 'email', $routeKey = 'email');
```

守护进程需要使用root用户运行，运行后会切换到普通用户，同时创建进程ID文件，以便进程停止的时候使用。

守护进程核心代码https://github.com/netkiller/SOA/blob/master/system/rabbitdaemon.class.php

6.2. 消息队列协议

消息协议是一个数组，将数组序列化或者转为JSON推送到消息队列服务器，这里使用json格式的协议。

```
$msg = array(
        'Namespace'=>'namespace',
        "Class"=>"Email",
        "Method"=>"smtp",
        "Param" => array(
                $mail, $subject, $message, null
        )
);
```

序列化后的协议

```
{"Namespace":"single","Class":"Email","Method":"smtp","Param":["netkiller@msn.com","Hello"," TestHelloWorld",null]}
```

使用json格式是考虑到通用性，这样推送端可以使用任何语言。如果不考虑兼容，建议使用二进制序列化，例如msgpack效率更好。

6.3. 消息队列处理

消息队列处理核心代码

https://github.com/netkiller/SOA/blob/master/system/rabbitmq.class.php

所以消息的处理在下面一段代码中进行

```
$this->queue->consume(function($envelope, $queue) {
```

```
        $speed = microtime(true);

        $msg = $envelope->getBody();
        $result = $this->loader($msg);
        $queue->ack($envelope->getDeliveryTag()); //手动发送ACK应答

        //$this->logging->info(''.$msg.' '.$result)
        $this->logging->debug('Protocol: '.$msg.' ');
        $this->logging->debug('Result: '. $result.' ');
        $this->logging->debug('Time: '. (microtime(true) - $speed) .'');
});
```

public function loader($msg = null) 负责拆解协议，然后载入对应的类文件，传递参数，运行方法，反馈结果。

Time 可以输出程序运行所花费的时间，对于后期优化十分有用。

提示

loader() 可以进一步优化，使用多线程每次调用loader将任务提交到线程池中，这样便可以多线程处理消息队列。

6.4. 测试

测试代码 https://github.com/netkiller/SOA/blob/master/test/queue/email.php

```
<?php
$queueName = 'example';
$exchangeName = 'email';
$routeKey = 'email';
$mail = $argv[1];
$subject = $argv[2];
$message = empty($argv[3]) ? 'Hello World!' : ' '.$argv[3];


$connection = new AMQPConnection(array(
        'host' => '192.168.4.1',
        'port' => '5672',
        'vhost' => '/',
        'login' => 'guest',
        'password' => 'guest'
        ));
$connection->connect() or die("Cannot connect to the broker!\n");

$channel = new AMQPChannel($connection);
$exchange = new AMQPExchange($channel);
$exchange->setName($exchangeName);
$queue = new AMQPQueue($channel);
$queue->setName($queueName);
$queue->setFlags(AMQP_DURABLE);
$queue->declareQueue();

$msg = array(
        'Namespace'=>'namespace',
        "Class"=>"Email",
        "Method"=>"smtp",
        "Param" => array(
                $mail, $subject, $message, null
        )
);

$exchange->publish(json_encode($msg), $routeKey);
printf("[x] Sent %s \r\n", json_encode($msg));


$connection->disconnect();
```

这里只给出了少量测试与演示程序，如有疑问请到读者群，或者公众号询问。

7. 多线程

上面消息队列 核心代码如下

```
$this->queue->consume(function($envelope, $queue) {
        $msg = $envelope->getBody();
        $result = $this->loader($msg);
        $queue->ack($envelope->getDeliveryTag());
});
```

这段代码生产环境使用了半年，发现效率比较低。有些业务场入队非常快，但处理起来所花的时间就比较长，容易出现队列堆积现象。

增加多线程可能更有效利用硬件资源，提高业务处理能力。代码如下

```
<?php
namespace framework;

require_once( __DIR__.'/autoload.class.php' );

class RabbitThread extends \Threaded {

        private $queue;
        public $classspath;
        protected $msg;

        public function __construct($queue, $logging, $msg) {
                $this->classspath = __DIR__.'/../queue';
                $this->msg = $msg;
                $this->logging = $logging;
                $this->queue = $queue;
        }
        public function run() {
                $speed = microtime(true);
                $result = $this->loader($this->msg);
                $this->logging->debug('Result: '. $result.' ');
                $this->logging->debug('Time: '. (microtime(true) - $speed) .'');
        }
        // private
        public  function loader($msg = null){

                $protocol       = json_decode($msg,true);
                $namespace      = $protocol['Namespace'];
                $class          = $protocol['Class'];
                $method         = $protocol['Method'];
                $param          = $protocol['Param'];
                $result         = null;

                $classspath = $this->classspath.'/'.$this->queue.'/'.$namespace.'/'.strtolower($class)  . '.class.php';
                if( is_file($classspath) ){
                        require_once($classspath);
                        //$class = ucfirst(substr($request_uri, strrpos($request_uri, '/')+1));
                        if (class_exists($class)) {
                                if(method_exists($class, $method)){
                                        $obj = new $class;
                                        if (!$param){
                                                $tmp = $obj->$method();
                                                $result = json_encode($tmp);
                                                $this->logging->info($class.'->'.$method.'()');
                                        }else{
                                                $tmp = call_user_func_array(array($obj, $method), $param);
                                                $result = (json_encode($tmp));
                                                $this->logging->info($class.'->'.$method.'("'.implode('","', $param).'")');
                                        }
                                }else{
                                        $this->logging->error('Object '. $class. '->' . $method. ' is not exist.');
                                }
                        }else{
                                $msg = sprintf("Object is not exist. (%s)", $class);
                                $this->logging->error($msg);
                        }
                }else{
                        $msg = sprintf("Cannot loading interface! (%s)", $classspath);
                        $this->logging->error($msg);
                }
                return $result;
        }
}

class RabbitMQ {

        const loop = 10;

        protected $queue;
        protected $pool;

        public function __construct($queueName = '', $exchangeName = '', $routeKey = '') {

                $this->config = new \framework\Config('rabbitmq.ini');
                $this->logfile = __DIR__.'/../log/rabbitmq.%s.log';
                $this->logqueue = __DIR__.'/../log/queue.%s.log';
                $this->logging = new \framework\log\Logging($this->logfile, $debug=true); //.H:i:s

                $this->queueName        = $queueName;
                $this->exchangeName     = $exchangeName;
                $this->routeKey         = $routeKey;

                $this->pool = new \Pool($this->config->get('pool')['thread']);

        }
        public function main(){

                $connection = new \AMQPConnection($this->config->get('rabbitmq'));
                try {
                        $connection->connect();
                        if (!$connection->isConnected()) {
                                $this->logging->exception("Cannot connect to the broker!".PHP_EOL);
                        }
                        $this->channel = new \AMQPChannel($connection);
                        $this->exchange = new \AMQPExchange($this->channel);
                        $this->exchange->setName($this->exchangeName);
```

```
                        $this->exchange->setType(AMQP_EX_TYPE_DIRECT); //direct类型
                        $this->exchange->setFlags(AMQP_DURABLE); //持久?
                        $this->exchange->declareExchange();
                        $this->queue = new \AMQPQueue($this->channel);
                        $this->queue->setName($this->queueName);
                        $this->queue->setFlags(AMQP_DURABLE); //持久?
                        $this->queue->declareQueue();

                        $this->queue->bind($this->exchangeName, $this->routeKey);

                        $this->queue->consume(function($envelope, $queue) {
                                $msg = $envelope->getBody();
                                $this->logging->debug('Protocol: '.$msg.' ');
                                //$result = $this->loader($msg);
                                $this->pool->submit(new RabbitThread($this->queueName, new \framework\log\Logging($this->logqueue, $debug=true),
                                $queue->ack($envelope->getDeliveryTag());
                        });
                        $this->channel->qos(0,1);
                }
                catch(\AMQPConnectionException $e){
                        $this->logging->exception($e->__toString());
                }
                catch(\Exception $e){
                        $this->logging->exception($e->__toString());
                        $connection->disconnect();
                        $this->pool->shutdown();
                }
        }
        private function fault($tag, $msg){
                $this->logging->exception($msg);
                throw new \Exception($tag.': '.$msg);
        }

        public function __destruct() {
        }
}
```

8. 订阅节点横向扩展

上面使用多线程达到对单机资源的充分使用，除此之外我们还可以横向扩展系统，增加订阅节点的数量。

9. 总结

该消息队列框架还比较简陋，但在生产环境已经运行很长一段时间，效果还是不错的。同时降低了消息队列的开发难度，开发者更多的时间是考虑业务逻辑的实现，而不用操心消息队列本身的使用。

10. 延伸阅读

PHP高级编程之守护进程

PHP高级编程之多线程