stone-docs

chefxu

Published
with GitBook

Table of Contents

关于Stone

Stone是一个优化Laravel框架性能的方案， 能大幅提升基于Laravel的程序性能，使Laravel能轻松应对高并发的使用场景。

优化原理

如果你正在考虑框架性能优化的问题， 你应该对PHP应该已经有足够的了解。 如你所知， PHP每次请求结束， 都会释放掉执行中建立的所有资源。这样有一个很大的好处：PHP程序员基本不用费力去考虑资源释放的问题，诸如内存，IO句柄，数据库连接等，请求结束时PHP将全部释放。PHP程序员几乎不用关心内存释放的问题，也很难写出内存泄露的程序。这让PHP变得更加简单容易上手， 直抒心意。但是同时也带来了一个坏处：PHP很难在请求间复用资源， 类似PHP框架这种耗时的工作， 每次请求都需要反复做——即使每次都在做同样的事情。也正因为如此，在PHP发展过程中，关于是否使用框架的争论也从未停止过。

Stone主要优化的就是这个问题。 在框架资源初始化结束后再开启一个FastCGI服务，这样， 新的请求过来是直接从资源初始化结束后的状态开始，避免每次请求去做资源初始化的事情。所以， 本质上， Stone运行时是常驻内存的，它和PHP-FPM一样，是一个FastCGI的实现，不同的是， FPM每次执行请求都需要重新初始化框架， Stone直接使用初始化的结果。

同样，事情总是有好有坏。使用Stone后的坏处是：PHP编程变得更难了。你需要考虑内存的释放，需要关心PHP如何使用内存。甚至， 你需要了解使用的框架，以免『不小心』写出让人『惊喜』的效果。同时， PHP的调试变得更难， 因为每次修改程序后需要重启进程才能看到效果。事实上，开发Stone时针对调试这方面做了不少工作。好处是：程序的性能得到极大的提高。

当然， 客观上的一些利好因素是：

- PHP的内存回收已经相当稳定和高效
- Swoole稳定性已经在相当多的项目中得到验证
- Laravel代码质量相当高

正是因为有了这些条件， 才使得Stone的出现成为可能， 感谢这个伟大的开源时代！

性能对比

应用类型	原始 **Laravel**	**Stone-Web**	**Stone-Server**	原生**php**直接 **echo**
laravel5 默认页面	150	3000	--	--
laravel5 简单接口	150	3000	8500	9500
laravel4 实际项目简单页面	70	1000	--	--
laravel4 简单接口	120	--	8200	9500
laravel4 实际项目首页	35	380	--	--

- 以上单位全部为RPS
- Stone相对于原始的Laravel有相当可观的提升
- 即使和一个简单的echo相比， Stone性能损失仅10%左右

测试环境如下：

```
PHP 5.6.17-0+deb8u1 (cli) (built: Jan 13 2016 09:10:12)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend
```

```
Linux office 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt20-1+deb8u3 (20
```

```
16核 Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
16G 内存
Laravel 4.2
Laravel 5.2
```

如果你对stone感兴趣， 进入下一章节了解如何5分钟快速使用Stone吧。

安装指南

版本选择

Stone推荐使用Laravel5， 但是也保持了对Laravel4的兼容性。 如果你是新项目，强烈建议使用Laravel5来开发。

运行模式选择

Stone提供两种运行模式：Stone-Server和Stone-Web。

1. Stone-Server

Stone-Server主要针对高性能要求的api调用场景。这种模式下，Session不会工作， 也不会执行Laravel MVC的流程，自然也不会执行Http中间件，所有请求直接交由RequestHandler处理。

一个最简单的Handler定义如下：

```php
<?php namespace App\Servers;

use Qufenqi\Stone\Contracts\RequestHandler;
use Response;

class Handler implements RequestHandler
{
    public function process()
    {
        return Response::make('hello, stone server!');
    }

    public function onWorkerStart()
    {

    }
}
```

如上，所有请求将直接给Handler的process方法处理，因此能得到最大化的性能。在测试中，相比原生php的echo输出，Stone-Server在加载全部框架资源后，性能损失不到10%。但是，你几乎可以直接使用原来Laravel写好的所有业务代码，使用Laravel的所有组件，也可以直接composer安装需要的组件。

关于Stone-Server的更详细介绍，请阅读运行模式章节

2. Stone-Web

Stone-Web的目的是优化现有的基于Laravel的网站项目。Stone接管Laravel的Http Kernel，负责协调MVC执行流程。在设计中，尽量遵循原来的流程。原来的过滤器，中间件都能正常工作。

为了设计Stone-Web，我阅读了大部分的Laravel代码，对Laravel 执行流程和设计架构有一定认识后才开始工作，以求尽可能的兼容原来的Laravel程序。但是，由于实际的项目代码可能很复杂，原来的程序没有考虑常驻内存的场景，加上我对Laravel的理解难免存在错误，因此在使用Stone-Web的时候一直小心翼翼。

在设计Stone的时候一直遵循着几个原则：

- 不破坏现有代码，保证现有代码100%能运行在php-fpm上

- 尽量提供保护机制， 降低产生意外错误的概率

这些都是为了尽量降低可能潜在的风险。

尽管如此，如果你对**PHP**内存使用没有足够的认识， 或者对你现在的项目质量没有足够的把握， 现阶段不推荐你把**Stone-Web**用于实际的生产环境。

关于Stone-Web的更详细介绍， 请阅读运行模式章节

了解风险

如果你对Stone已经有了初步了解， 并愿意尝试， 请继续阅读风险提示。

风险提示

使用Stone的一个很重要的事情是， 需要始终对内存使用抱有敬畏之心。 在未充分了解风险前， 请不要在实际项目中轻易尝试，否则可能产生非常严重的后果！！

比如，Laravle的Cookie实现了单例模式：

```
<?php

namespace Illuminate\Cookie;

use Illuminate\Support\ServiceProvider;

class CookieServiceProvider extends ServiceProvider
{
    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton('cookie', function ($app) {
            $config = $app['config']['session'];

            return (new CookieJar)->setDefaultPathAndDomain($config
        });
    }
}
```

这样，开发者在任何地方可以往Response里追加cookie， 使用：

```
Cookie::queue('key', 'value', 60);
```

所有的Cookie都被维护在一个数组中，等待Response发送到客户端：

```php
<?php

namespace Illuminate\Cookie;

use Illuminate\Support\Arr;
use Symfony\Component\HttpFoundation\Cookie;
use Illuminate\Contracts\Cookie\QueueingFactory as JarContract;

class CookieJar implements JarContract
{
    /**
     * The default path (if specified).
     *
     * @var string
     */
    protected $path = '/';

    /**
     * The default domain (if specified).
     *
     * @var string
     */
    protected $domain = null;

    /**
     * The default secure setting (defaults to false).
     *
     * @var bool
     */
    protected $secure = false;

    /**
     * All of the cookies queued for sending.
     *
     * @var array
     */
    protected $queued = [];
```

在原来的PHP-FPM下，这不会有任何问题，因为请求结束后PHP对释放掉所有资源。但Stone常驻内存后，请求的资源没有在请求结束后释放，因为cookie对象被ioc容器引用，PHP GC不会回收这部分内存。因此下一个请求再使用**Cookie**的时候获取的是同一个**Cookie**对象。

比如，第一个请求：

```
Cookie::queue('first', 'test', 60);
```

这个时候queue数组里已经有first的cookie。

第二个请求：

```
Cookie::queue('secend', 'test', 60);
```

由于获取的还是上一个cookie对象，因此queue数组包含first和second的cookie！

想像下，如果这是一个关于Session Id的cookie，会产生多么严重的后果。

同样，**Laravel**的**Auth**对象，也可能出现类似的问题。

内存泄露什么的可能只是影响稳定性，但是这个处理不当会带来难以估计的损失，这是使用Stone或者类似常驻内存方案的最大风险。

Stone通过实例快照来解决这个问题。简单来说就是在对象建立的时候把对象的当前状态保存下来，当请求结束后通过保存的数据把实例恢复成初始化的状态，这样就可以避免这个问题。

在config/stone.php的配置中，你可以定义哪些对象需要建立快照。

```
        // 需要建立快照的绑定
        'snap_bindings' => [
            'view',
            'cookie',
            'session',
            'session.store',
            //'config', // debugbar 需要重置config
        ],
```

有关资源复用的更多细节，请查看理解资源复用这个章节。

开始使用Stone

如果你已经了解这样的潜在风险，并对此信心满满，那开始使用吧：

- 在Laravel5中使用Stone
- 在Laravel4中使用Stone

Laravel5中使用Stone

Laravel5中安装Stone

1. 安装依赖包，如果你只运行Stone-Server， 则runkit可以不安装。

```
sudo pcel install swoole
sudo pcel install runkit
```

2. composer安装Stone

```
composer require qufenqi/stone:dev-master
```

3. 修改config/app.php, 加载Stone的Service Provider,

```
'providers' => [
    // laravel定义的provider
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    ....
    .... // 中间省略的其他provider
    ....
    Qufenqi\Stone\StoneServiceProvider::class,

    // 应用层定义的provider
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

],
```

4. 配置Stone: 新建 config/stone.php

```
return [

    // server模式配置
    'server' => [
        'handler' => 'App\Servers\Handler', // request handler
        'user' => 'apple', // run user
        'group' => 'apple', // run group
        'domain' => '/var/run/stone-server-fpm.sock',
        'pid' => '/run/stone-fpm.pid',
        'process_name' => 'stone-server-fpm',
        'worker_num' => 30,
    ],

    // web模式配置
    'web' => [
        'user' => 'apple', // run user
        'group' => 'apple', // run group
        'domain' => '/var/run/stone-web-fpm.sock', // unix dom
        'pid' => '/run/stone-web.pid',
        'process_name' => 'stone-web-server',
        'worker_num' => 30,

        // 需要建立快照的绑定
        'snap_bindings' => [
            'view',
            'cookie',
            'session',
            'session.store',
            //'config', // debugbar 需要重置config
        ],

    ],
];
```

5. 安装完成

Laravel5中设置Web运行模式

1. 修改app/Http/Kernel.php, 让Stone的Kernel接管请求的处理。

```
// 根据当前的运行sapi决定使用哪个kernel来处理请求， 这样FPM和Stone可以
if (php_sapi_name() == 'cli') {
    class BaseKernel extends StoneKernel {}
} else {
    class BaseKernel extends HttpKernel {}
}

class Kernel extends BaseKernel
```

2. 运行Stone-Web, Web模式处在开发阶段， 所以默认不会以deamon模式启动， 便于调试

```
sudo php ./public/index.php
```

3. 修改nginx配置

```
location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;
    # fastcgi_pass unix:/var/run/php5-fpm.sock; # PHP-FPM
    fastcgi_pass unix:/var/run/stone-web-fpm.sock; # Stone
    include fastcgi_params;
}
```

```
sudo nginx -s reload
```

4. 完成

Laravel5中设置Server运行模式

1. 修改app\Console\Kernel.php

```
protected $commands = [
    // Commands\Inspire::class,
    \Qufenqi\Stone\Console\Commands\StoneServer::class, // 添加
];
```

2. 定义请求处理类， 我定义在app\Servers\Handler.php

> 注意 这个其实就是 stone.php 配置里的 server.handler

```
<?php namespace App\Servers;

use Qufenqi\Stone\Contracts\RequestHandler;
use Response;

class Handler implements RequestHandler
{
    public function process()
    {
        return Response::make('hello, stone server!');
    }

    public function onWorkerStart()
    {

    }
}
```

3. 运行Stone-Server

```
sudo php ./artisan stone:server
```

4. 修改nginx配置

```
location /server/ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;
    fastcgi_pass unix:/var/run/stone-server-fpm.sock; # Stone
    include fastcgi_params;
}
```

```
sudo nginx -s reload
```

5. 完成

Laravel4中使用Stone

Laravel4中安装Stone

1. 安装依赖包, 如果你需要Stone-Server，则不需要安装runkit。

```
sudo pcel install swoole
sudo pcel install runkit
```

2. composer安装Stone

```
composer require qufenqi/stone:dev-laravel-4.x
```

3. 修改app/config/app.php, 加载Stone的Service Provider,

```
'providers' => array(

        'Illuminate\Foundation\Providers\ArtisanServiceProvide
        'Illuminate\Auth\AuthServiceProvider',
        'Illuminate\Cache\CacheServiceProvider',
        ...
        ...
        ...
        'Qufenqi\Stone\StoneServiceProvider',
),
```

4. 配置Stone: 新建 app/config/stone.php

```
return [

    // server模式配置
    'server' => [
        'handler' => 'App\Servers\Handler', // request handler
        'user' => 'apple', // run user
        'group' => 'apple', // run group
        'domain' => '/var/run/stone-server-fpm.sock',
        'pid' => '/run/stone-fpm.pid',
        'process_name' => 'stone-server-fpm',
        'worker_num' => 30,
    ],

    // web模式配置
    'web' => [
        'user' => 'apple', // run user
        'group' => 'apple', // run group
        'domain' => '/var/run/stone-web-fpm.sock', // unix dom
        'pid' => '/run/stone-web.pid',
        'process_name' => 'stone-web-server',
        'worker_num' => 30,

        // 需要建立快照的绑定
        'snap_bindings' => [
            'view',
            'cookie',
            'session',
            'session.store',
            //'config', // debugbar 需要重置config
        ],

    ],
];
```

Laravel4中设置Web运行模式

1. 修改public/index.php与bootstrap/start.php, 让Stone的Kernel接管请求的处理。

```
// 修改public/index.php
if (PHP_SAPI == 'cli') {
    define('STONE_WEB_MODE', true);
    $_SERVER['RUNENV'] = 'local';
}
```

```
// 修改bootstrap/start.php
if (defined('STONE_WEB_MODE')) {
    $app = new Qufenqi\Stone\Foundation\Application;
} else {
    $app = new Illuminate\Foundation\Application;
}
```

2. 运行Stone-Web, Web模式处在开发阶段， 所以默认不会以deamon模式启动， 便于调试

```
sudo php ./public/index.php
```

3. 修改nginx配置

```
location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;
    # fastcgi_pass unix:/var/run/php5-fpm.sock; # PHP-FPM
    fastcgi_pass unix:/var/run/stone-web-fpm.sock; # Stone
    include fastcgi_params;
}
```

```
sudo nginx -s reload
```

4. 完成

Laravel4中设置Server运行模式

1. 修改app\start\artisan.php

```
Artisan::add(new Qufenqi\Stone\Console\Commands\StoneServer);
```

2. 定义请求处理类，我定义在app\Servers\Handler.php

> 注意 这个其实就是 stone.php 配置里的 server.handler

```
<?php namespace App\Servers;

use Qufenqi\Stone\Contracts\RequestHandler;
use Response;

class Handler implements RequestHandler
{
    public function process()
    {
        return Response::make('hello, stone server!');
    }

    public function onWorkerStart()
    {

    }
}
```

3. 运行Stone-Server

```
sudo php ./artisan stone:server
```

4. 修改nginx配置

```
location /server/ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;
    fastcgi_pass unix:/var/run/stone-server-fpm.sock; # Stone
    include fastcgi_params;
}
```

```
sudo nginx -s reload
```

5. 完成

系统架构

理解资源复用

PHP资源回收

在Stone中资源可以不在请求结束后销毁，也可以在结束后自行销毁，这取决于你如何使用这个资源。和C语言不一样，PHP有基于引用计数的内存回收机制，当一个对象没有被引用时，对象就会被回收，并不需要你手动free。

我们写一个小例子确认一下：

首先，我们临时调低一下linux一个进程能打开的最大文件数：

```
ulimit -n 8
```

设置一个进程最大打开文件句柄数为8后，我们写这样一段php程序，保存为/tmp/ulimit.php：

```
<?php

$i = 0;
while ($i < 10) {
    $fp = fopen('/tmp/file' . $i, 'w');
    $i++;
}

echo "ok\n";
```

运行这个php程序，输出『ok』：

```
php /tmp/ulimit.php
ok
```

然后，我们对这个程序做一个小的修改，把原来的$fp变成一个数组:

```php
<?php

$fp = [];
$i = 0;
while ($i < 10) {
    $fp[] = fopen('/tmp/file' . $i, 'w');
    $i++;
}

echo "ok\n";
```

再运行就得到这样的效果：

```
PHP Warning:  fopen(/tmp/file4): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file5): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file6): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file7): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file8): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file9): failed to open stream: Too many op
ok
```

这证明了之前关于资源回收的观点，第一个程序，fp不断被新的句柄覆盖，旧的句柄没有被引用，php自动回收了这个资源，即使我们并没有手动去fclose它。第二个程序中，由于文件句柄被数组持有，所以php无法回收，导致超出了进程最大打开文件数。

> 细心的朋友可能会问，明明最大文件数是8，怎么从file4就开始报错了，这是因为在linux中，每一个进程都会自动打开3个文件句柄：输入，输出，错误。同时，php执行ulimit.php文件，也占用了一个句柄，再加上file0-file3, 刚好8个文件句柄。

Server模式为什么相对安全？

再回头看Server模式，核心逻辑都是一个process方法发起的，process结束后，方法里申请的资源就会被释放。

```php
<?php
class Handler implements RequestHandler
{
    private $data;

    public function process()
    {
        $fp = fopen('/tmp/123', 'w');
        $db = new PDO('mysql://xxxx');
        $obj = new stdclass;
    }

    public function onWorkerStart()
    {
        $this->data = 'xxxxx';
    }
}
```

如上面代码, $fp, $db, $obj在请求结束后就会被销毁， 但是$data不会， 因为$data被handler引用了， 而handler常驻在内存中。

之所以说Server模式『相对安全』，就是因为根据我们大多数程序员的使用习惯，我们都是在运行时通过new的方式来创建对象，这样的做法在程序结束之后自然而然地被释放了。

Server模式真的一定安全吗？

先看下面这段代码， 我们模拟一下Server的执行， process方法里打开一些文件。

```php
<?php
class Container
{
    private $fp = [];

    public function addFp($fp)
    {
        $this->fp[] = $fp;
    }
}

function process($i)
{
    $container = new Container;
    $container->addFp(fopen('/tmp/file' . $i, 'w'));
}

$i = 0;
while ($i < 10) {
    process($i);
    $i++;
}

echo "ok\n";
```

执行后正常输出ok，这是因为fp虽然被container持有，但是process结束后，container被自动释放，由于fp被引用的container已经被释放，再也没有被其他任何对象引用，所以fp自然也就释放了。

```
ok
```

我们稍微修改一下程序，把Container的方法设定为静态方法，属性设置为静态属性，再测试一下：

```php
<?php
class Container
{
    private static $fp = [];

    public static function addFp($fp)
    {
        self::$fp[] = $fp;
    }
}

function process($i)
{
    Container::addFp(fopen('/tmp/file' . $i, 'w'));
}

$i = 0;
while ($i < 10) {
    process($i);
    $i++;
}

echo "ok\n";
```

这个时候输出如下：

```
PHP Warning:  fopen(/tmp/file4): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file5): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file6): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file7): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file8): failed to open stream: Too many op
PHP Warning:  fopen(/tmp/file9): failed to open stream: Too many op
ok
```

原因是php中类是全局存在的，始终有一个global的域引用着。PHP中，静态变量被定义在类上，所以这个变量不会销毁。那变量引用的文件句柄fp自然也不会销毁，即使函数已经执行完毕。

温习下Laravel的控制反转

Laravel这个框架被设计得高度灵活， 整个框架的基于IOC容器建立的，应用启动后，app就是一个大的容器。我们简单分析下这行代码发生了什么， 为了简单， 我忽略了实现细节：

```
$user = Auth::user();
```

这是一段使用Laravel几乎每个人都会用到的代码，那这行代码发生了什么呢？

Auth是一个Facade类的别名：

```
'Auth' => Illuminate\Support\Facades\Auth::class,
```

顺藤摸瓜，Facade最终实例化Auth对象其实就是通过app实例化的：

```
    public static function __callStatic($method, $args)
     {
         $instance = static::getFacadeRoot();

         if (! $instance) {
             throw new RuntimeException('A facade root has not been
         }

         switch (count($args)) {
             case 0:
                 return $instance->$method();

             case 1:
                 return $instance->$method($args[0]);

             case 2:
                 return $instance->$method($args[0], $args[1]);

             case 3:
                 return $instance->$method($args[0], $args[1], $args

             case 4:
                 return $instance->$method($args[0], $args[1], $args

             default:
                 return call_user_func_array([$instance, $method], $
         }
     }
```

使用'auth'这个绑定：

```
    protected static function getFacadeAccessor()
    {
        return 'auth';
    }
```

'auth'? app怎么知道实例化哪个对象呢？其实在app启动时， 就通过ServiceProvider注册了：

```
    protected function registerAuthenticator()
    {
        $this->app->singleton('auth', function ($app) {
            // Once the authentication service has actually been re
            // we will set a variable in the application indicating
            // know that we need to set any queued cookies in the a
            $app['auth.loaded'] = true;

            return new AuthManager($app);
        });

        $this->app->singleton('auth.driver', function ($app) {
            return $app['auth']->guard();
        });
    }
```

这样，一个控制反转就完成了。 饶了这么大一圈， 简单的说就是：

产品需求： 世界和平

开发一:

```
$person = new SuperMan();
$person->killAllEnemies();
```

开发二：

```
$person = App::make('power');
$person->killAllEnemies();
```

开发一直接实例化一个超人， 控制权在调用者， 调用者决定自己具体需要什么。 当某一天超人不在， 或者超人打不过敌人的时候， 需求就挂了。

开发二没有直接决定自己具体需要什么，他只是告诉提供者，我需要一个具有超能力的对象（面向接口编程），这样，控制权交还给提供者，提供者发现超人不行的时候，就派出超级赛亚人了 :)

控制反转后，提供者与调用者不再是强依赖的关系，大大提高了系统的灵活性。有点跑题了，关于控制反转的话题，有机会再单独聊。

我们的结论是：

```
$user = Auth::user();
```

这样一行代码，Auth对象被ioc容器引用，被Facade类引用（为了避免反复触发魔术方法，提高效率），所以，函数结束后，这个对象并不会被回收。

因此，Server模式下，也不一定安全。

Stone如何解决这个问题

Auth对象之所以不被释放，是因为请求结束后仍然被APP和Facade引用，所以只需要解除引用就可以了。但是新的问题又来了：Stone怎么知道哪些实例需要释放，哪些实例不需要释放呢？

其实Stone并没有什么魔法，它释放实例基于一个很简单的原理，在app加载完成后，将app建立快照，请求结束后再通过快照恢复app，这样在请求时新建立的实例引用就会被解除。

比如：app初始化后，容器里的实例有cookie，auth等，把当前实例列表保存下来。请求中又新注册绑定了一些新的实例，容器里的实例又增加了instance1，instance2等。请求结束后，实例列表恢复成初始状态，这样app对于instance1，instance2的引用就解除了。

在Stone中的做法：

请求开始之前：

```
            $this->boot();
            $this->injectInstance();
            $this->snapApp();
```

请求结束之后：

```
            $response = with($stack = $this->getStackedClient())->h
            $stack->terminate($request, $response);
            $this->restoreApp();
```

Facade的实现类似。

但是，如果你在请求期间定义静态属性来保存引用，这种情况下是没法自动释放的。而且，我们也无法确定开发者到底是否需要自动释放。如果你能理解这些，利用好Stone资源复用的能力，一定可以写出非常高效的程序，反之，如果你不能很好理解，可能等待你的是一个个大坑。

这都取决于我们自己，不是吗？

运行模式

Stone提供两种运行模式： Stone-Server和Stone-Web。

Stone-Server

Server模式的请求处理的Kernel是：App\Console\Kernel， 执行流程与执行别的artisan命令一致。因此，process方法相当于在编写一个artisan命令的fire方法。你可以使用原来在artisan命令里可以使用的所有组件和功能。

Stone-Server执行堆栈：

```
Stack trace:
#0 vendor/qufenqi/stone/src/Qufenqi/Stone/Console/Commands/StoneSer
#1 [internal function]: Qufenqi\Stone\Console\Commands\StoneServer-
#2 vendor/laravel/framework/src/Illuminate/Container/Container.php(
#3 vendor/laravel/framework/src/Illuminate/Console/Command.php(169)
#4 vendor/symfony/console/Command/Command.php(256): Illuminate\Cons
#5 vendor/laravel/framework/src/Illuminate/Console/Command.php(155)
#6 vendor/symfony/console/Application.php(791): Illuminate\Console\
#7 vendor/symfony/console/Application.php(186): Symfony\Component\C
#8 vendor/symfony/console/Application.php(117): Symfony\Component\C
#9 vendor/laravel/framework/src/Illuminate/Foundation/Console/Kerne
#10 artisan(36): Illuminate\Foundation\Console\Kernel->handle(Objec
#11 {main}
```

Stone-Web

Stone-Web相对复杂， 处理请求的Kernel是App\Http\Kernel, 这个Kernel继承自Qufenqi\Stone\Foundation\Http， 接管了原来的Laravel的Kernel。

这样的主要目的是：

- 细化ServiceProvider为Boot类型和Request类型

  细节请查看：服务提供者

- 给予App和Facade快照恢复的能力

  细节请查看：理解资源复用

- 改变系统某些ServiceProvider的默认行为

  比如SessionServiceProvider运行在php cli模式下时会把driver设置成array，这样Session没法保存， 需要做出相应处理。

目前还没有找到完美实现Stone-Web的方案， 因此关于Stone-Web的实现， 可能会随时调整， 请知悉。 对于Stone-Web的实现过程感兴趣可以查看：分析，理解，优化LARAVEL

服务提供者

Laravel是基于IOC容器建立的，所以服务提供者随处可见。一般来说，在Laravel中，服务提供者会做两个事情：register和boot。register往容器里注册对象，boot在系统初始化的时候做一些事情。在Laravel中这些被统一称作ServiceProvider。

而在Stone中，Service Provider被分成两种类型: Boot Service Provider 和 Request Service Provider。主要原因是，Service Provider只会在app初始化的时候执行，使用Stone后，多次请求只会执行一次，对于某些Service Provider会存在问题。

所以被拆开为：

- Boot Service Provider，仅在进程初始化时执行一次
- Request Service Provider，每次请求都会执行

如何定义服务提供者

Stone设计一直有个原则，100%兼容原有PHP-FPM体系。所以，定义在config/app.php中的providers，默认都会是boot类型的。

```
    'providers' => [

        /*
         * Laravel Framework Service Providers...
         */
        Illuminate\Auth\AuthServiceProvider::class,
        Illuminate\Broadcasting\BroadcastServiceProvider::class,
        Illuminate\Bus\BusServiceProvider::class,
        Illuminate\Cache\CacheServiceProvider::class,
        Illuminate\Foundation\Providers\ConsoleSupportServiceProvic
        Illuminate\Cookie\CookieServiceProvider::class,
        Illuminate\Database\DatabaseServiceProvider::class,
        Illuminate\Encryption\EncryptionServiceProvider::class,
        Illuminate\Filesystem\FilesystemServiceProvider::class,
        Illuminate\Foundation\Providers\FoundationServiceProvider::
        Illuminate\Hashing\HashServiceProvider::class,
        Illuminate\Mail\MailServiceProvider::class,
        Illuminate\Pagination\PaginationServiceProvider::class,
        Illuminate\Pipeline\PipelineServiceProvider::class,
        Illuminate\Queue\QueueServiceProvider::class,
        Illuminate\Redis\RedisServiceProvider::class,
        Illuminate\Auth\Passwords\PasswordResetServiceProvider::cla
        Illuminate\Session\SessionServiceProvider::class,
        Illuminate\Translation\TranslationServiceProvider::class,
        Illuminate\Validation\ValidationServiceProvider::class,
        Illuminate\View\ViewServiceProvider::class,
        Qufenqi\Stone\StoneServiceProvider::class,

        /*
         * Application Service Providers...
         */
        App\Providers\AppServiceProvider::class,
        App\Providers\AuthServiceProvider::class,
        App\Providers\EventServiceProvider::class,
        App\Providers\RouteServiceProvider::class,
    ],
```

除非你在app/stone.php里重新定义为request类型：

```
    'web' => [
        'user' => 'apple', // run user
        'group' => 'apple', // run group
        'domain' => '/var/run/stone-web-fpm.sock',

        'request-providers' => [
            Illuminate\Cookie\CookieServiceProvider::class,
        ],
    ],
```

这样的目的还是为了PHP-FPM，原来的providers不需要做修改，这样使用PHP-FPM时不会有任何问题。

例子：安装debugbar

通过对debugbar的使用，更深入了解两种类型的区别。

实例快照

快照是为了解决请求结束后实例状态恢复的问题， 具体原因请查看： 理解资源复用 从实现上说， Stone的快照实现分为三种: Facade的实现， App的实现， Instance的实现。

Facade快照的实现

App容器快照的实现

组件Instance快照的实现

使用指南

使用Web模式

使用Server模式

实践建议

结尾

反馈与建议

常见问题

完善中

联系我们

- https://github.com/chefxu/stone
- 邮件：rssidea(at)qq.com

致谢

感谢这个开源时代：

- http://php.net/
- http://www.swoole.com/
- http://php.net/runkit
- http://www.qufenqi.com