<https://learnku.com/docs/laravel-core-concept/5.5>

# 依赖注入、控制翻转、反射

# 反射

根据类名返回该类的任何信息，比如该类有什么方法，参数，变量等等

// 获取User的reflectionClass对象

$reflector = new reflectionClass(User::class);

// 拿到User的构造函数

$constructor = $reflector->getConstructor();

// 拿到User的构造函数的所有依赖参数

$dependencies = $constructor->getParameters();

getParameters此方法会返回一个数组，包含方法每个参数的 ReflectionParameter 对象

// 创建user对象

$user = $reflector->newInstance();

// 创建user对象，需要传递参数的

$user = $reflector->newInstanceArgs($dependencies = []);

**laraval的源码使用大量的闭包和回调，所以在学习其源码之前要先理解闭包和回调**

https://www.cnblogs.com/bndong/p/7988365.html

# Closure匿名函数、闭包

匿名函数：没有名字的函数，最经常用作回调函数的参数值

## 1.匿名函数赋值

$a=function($name){

echo '1231'.$name;

};

//调用闭包函数

$a('123');

### 在函数中定义闭包函数并调用

//在函数中定义闭包函数并调用

function test(){

$fun=function($str){

echo $str;

};

$fun('aaaa');

}

test();//输出aaaa

### 在函数中把匿名函数返回，并调用它

// 在函数中把匿名函数返回，并调用它

function test(){

return function($str){

echo $str;

};

}

$fun=test();

$fun('aaaa');

### 匿名函数当做参数传递，并且调用它

// 把匿名函数当做参数传递，并且调用它

function test($fun){

$fun('aaaa');

}

$printStr=function($str){

echo $str;

};

test($printStr);//输出aaaa

## 2.use关键字用于连接闭包和外界变量

use关键字用于连接闭包和外界变量

function getMoney() {

$rmb = 1;

$dollar = 8;

$func = function() use ( $rmb ) {

echo $rmb;

echo $dollar;

};

$func();

}

getMoney();

//输出：1

dollar没有在use关键字中声明，在这个匿名函数里也就不能获取到它

**use引用的变量是一个副本，在匿名函数中修改引用的变量，原来的外界变量是不变的**

function getMoney() {

$rmb = 1;

$func = function() use ( $rmb ) {

echo $rmb.'<br>';

//把$rmb的值加1

$rmb++;

};

$func();

echo $rmb;

}

getMoney();

//输出：

//1

//1

可以用完全引用来改变外界变量

function getMoney() {

$rmb = 1;

$func = function() use ( &$rmb ) {

echo $rmb.'<br>';

//把$rmb的值加1

$rmb++;

};

$func();

echo $rmb;

}

getMoney();

//输出：

//1

//2

## 3.闭包绑定

通过bindTo将闭包内部状态绑定到对象上，

class A {

private static $sfoo = 1;

private $ifoo = 2;

}

$cl1 = static function() {

return A::$sfoo;

};

$cl2 = function() {

//匿名函数中莫名奇妙的有个this,这个this关键词

//就是说明这个匿名函数是需要绑定在类中的。

return $this->ifoo;

};

$bcl1 = Closure::bind($cl1, null, 'A');

$bcl2 = Closure::bind($cl2, new A(), 'A');

echo $bcl1(), "\n";//输出 1

echo $bcl2(), "\n";//输出 2

## 闭包执行

Function test(Closure $closure){

Return $closure();//执行闭包 并不是Closure()

}

<https://www.cnblogs.com/phpper/p/6993093.html>

**源码位置：vendor/laravel/illuminate**

## 4.闭包传参

*//闭包传参*$a=**function**(){  
 **return function** ($b,$c){  
 **echo'123'**;  
 };  
};  
$b=$a();  
$b(1,2);

## 5.闭包嵌套传参

*//闭包嵌套传参*$a=**function** ($x,$y){  
 **echo'123'**.$x.$y;  
};  
$b=**function** ($x,$y)**use** ($a){  
 **return** $a;  
};  
$c=$b(1,1);  
$c(**'ee'**,**'rr'**);  
*//输出123eerr*

## 6.方法中返回闭包

*//在方法中返回闭包 不用在方法中设置传递参数   
//function test($x,$y,$z)***function** test(){  
 **return function** ($x,$y,$z){  
 **echo '12132'**.$x.$y.$z;  
 };  
}  
$a=test();  
$a(**'e'**,**'r'**,**'t'**);

# Laravel启动流程

## 1.源码解析

参考：<https://blog.csdn.net/kwinH/article/details/56284633?utm_source=blogxgwz9>

Index.php

源码：

**<?php***/\*\*  
 \* Laravel - A PHP Framework For Web Artisans  
 \*  
 \** ***@package*** *Laravel  
 \** ***@author*** *Taylor Otwell <****taylor@laravel.com****>  
 \*/  
  
define*(**'LARAVEL\_START'**, *microtime*(**true**));**require *\_\_DIR\_\_***.**'/../vendor/autoload.php'**; *//加载运行应用程序，并向浏览器返回请求*$app = **require\_once *\_\_DIR\_\_***.**'/../bootstrap/app.php'**;  
  
 *//获取App\Http\Kernel 实例*$kernel = $app->make(Illuminate\Contracts\Http\Kernel::***class***);  
  
*//*$response = $kernel->handle(  
 $request = Illuminate\Http\Request::*capture*()  
);  
  
$response->send();  
  
$kernel->terminate($request, $response);

### require \_\_DIR\_\_.'/../vendor/autoload.php';

引入composer提供的类自动加载器

### 2、require\_once \_\_DIR\_\_.'/../bootstrap/app.php';

加载应用程序

#### 源码

**<?php**$app = **new** Illuminate\Foundation\Application(  
 $\_ENV[**'APP\_BASE\_PATH'**] ?? *dirname*(***\_\_DIR\_\_***)  
);  
  
*/\*  
  
| 注册三个单例到ioc容器中  
|分别是 http处理核心类（App\Http\Kernel）,cli处理核心类（App\Console\Kernel）,和异常处理（App\Exceptions\Handler）.  
\*/*$app->singleton(  
 Illuminate\Contracts\Http\Kernel::***class***,  
 App\Http\Kernel::***class***);  
  
$app->singleton(  
 Illuminate\Contracts\Console\Kernel::***class***,  
 App\Console\Kernel::***class***);  
  
$app->singleton(  
 Illuminate\Contracts\Debug\ExceptionHandler::***class***,  
 App\Exceptions\Handler::***class***);**return** $app;

#### 2.1$app = new Illuminate\Foundation\Application

**源码主要部分：**

**class** Application **extends** Container **implements** ApplicationContract, HttpKernelInterface

Application继承了**use** Illuminate\Container\Container;实现ApplicationContract, HttpKernelInterface两个接口

而

**use** Illuminate\Container\Container;

实现了**class** Container **implements** ArrayAccess,

**use** Illuminate\Contracts\Container\Container

接口

ArrayAccess是什么下面介绍

**public function** \_\_construct($basePath = **null**)  
{  
 *//为应用程序设置基础路径* **if** ($basePath) {  
 *// 在容器中绑定所有的应用程序路径  
 //如config public 等* $this->setBasePath($basePath);  
 }  
 *//绑定Appliation和Container到instance容器* $this->registerBaseBindings();  
 *//将所有的服务注册到容器中* $this->registerBaseServiceProviders();  
 *//将核心类的别名注册到容器中 即$alias=[]数组* $this->registerCoreContainerAliases();  
}

##### 2.11$this->setBasePath

*//为应用程序设置基础路径*

**源码**

**public function** setBasePath($basePath)  
{  
 $this->**basePath** = *rtrim*($basePath, **'\/'**);  
 *// 在容器中绑定所有的应用程序路径* $this->bindPathsInContainer();  
  
 **return** $this;  
}

------

**protected function** bindPathsInContainer()  
{  
 *//instance 将$abstract,$instance以key,value形式保存在Application->instances[]数组里* $this->instance(**'path'**, $this->path());  
 $this->instance(**'path.base'**, $this->basePath());  
 $this->instance(**'path.lang'**, $this->langPath());  
 $this->instance(**'path.config'**, $this->configPath());  
 $this->instance(**'path.public'**, $this->publicPath());  
 $this->instance(**'path.storage'**, $this->storagePath());  
 $this->instance(**'path.database'**, $this->databasePath());  
 $this->instance(**'path.resources'**, $this->resourcePath());  
 $this->instance(**'path.bootstrap'**, $this->bootstrapPath());  
}

**//instance即绑定一个实例 具体的分析 在下面的ioc绑定instance**

**public function** instance($abstract, $instance)  
 {  
 *//从上下文绑定别名缓存中删除别名  
// 此方法完成工作：如果$this->aliases[$searched]即要绑定实例的别名不存在 return  
// 否则删除abstractAliases数组中相应的值，unset($this->abstractAliases[$abstract][$index]);* $this->removeAbstractAlias($abstract);  
 *//确定给定的抽象类型是否已绑定* $isBound = $this->bound($abstract);  
 *//刪除 已注册的类型别名数组的值  
 //unset() 删除不存在变量 不报错* **unset**($this->**aliases**[$abstract]);  
  
*// We'll check to determine if this type has been bound before, and if it has  
// we will fire the rebound callbacks registered with the container and it  
// can be updated with consuming classes that have gotten resolved here.  
 //我们将检查以确定这种类型以前是否被绑定过，  
 //如果有，我们将触发注册到容器的反弹回调函数，它可以用在这里已经解决的消费类进行更新。* $this->**instances**[$abstract] = $instance;  
 *//如果别名已绑定* **if** ($isBound) {  
 $this->rebound($abstract);  
 }  
  
 **return** $instance;  
 }

##### 2.12$this->registerBaseBindings();

*//绑定Appliation和Container到instance容器*

源码：

**protected function** registerBaseBindings()  
{  
 *//设置容器的共享实例  
 //return static::$instance = $container;  
 //即将当前的instance设成当前容器实例* **static**::*setInstance*($this);  
 *//将容器以key,value形式保存在Application->instances[]数组里 key为app* $this->instance(**'app'**, $this);  
  
 $this->instance(Container::***class***, $this);  
 $this->singleton(Mix::***class***);  
 *//框架的各目录路径* $this->instance(PackageManifest::***class***, **new** PackageManifest(  
 **new** Filesystem, $this->basePath(), $this->getCachedPackagesPath()  
 ));  
}

----

// PackageManifest的构造函数，主要完成下面几个的初始化

**public function** \_\_construct(Filesystem $files, $basePath, $manifestPath)  
{  
 $this->**files** = $files;*//文件对象* $this->**basePath** = $basePath;*//框架路径* $this->**manifestPath** = $manifestPath;*//bootstarp/cache/packages.php文件* $this->**vendorPath** = $basePath.**'/vendor'**;  
}

##### 2.13$this->registerBaseServiceProviders();

*//将所有的服务注册到容器中*

**源码**

**protected function** registerBaseServiceProviders()  
{  
 $this->register(**new** EventServiceProvider($this));  
 $this->register(**new** LogServiceProvider($this));  
 $this->register(**new** RoutingServiceProvider($this));  
}

---

**Register源码**

*/\*\*  
 \* 向应用程序注册服务提供商  
 \*/***public function** register($provider, $force = **false**)  
{  
 *//功能是检测$provider是否已经保存在$this->serviceProviders数组里了* **if** (($registered = $this->getProvider($provider)) && ! $force) {  
 **return** $registered;  
 }  
  
 *// If the given "provider" is a string, we will resolve it, passing in the  
 // application instance automatically for the developer. This is simply  
 // a more convenient way of specifying your service provider classes.* **if** (*is\_string*($provider)) {  
 $provider = $this->resolveProvider($provider);  
 }  
 *//运行服务的注册方法* $provider->register();  
  
 *// If there are bindings / singletons set as properties on the provider we  
 // will spin through them and register them with the application, which  
 // serves as a convenience layer while registering a lot of bindings.* **if** (*property\_exists*($provider, **'bindings'**)) {  
 **foreach** ($provider->**bindings as** $key => $value) {  
 $this->bind($key, $value);  
 }  
 }  
  
 **if** (*property\_exists*($provider, **'singletons'**)) {  
 **foreach** ($provider->**singletons as** $key => $value) {  
 $this->singleton($key, $value);  
 }  
 }  
 *//将给定的服务提供者标记为已注册* $this->markAsRegistered($provider);  
  
 *// If the application has already booted, we will call this boot method on  
 // the provider class so it has an opportunity to do its boot logic and  
 // will be ready for any usage by this developer's application logic.  
 //指明程序是由已启动 未启动为false* **if** ($this->isBooted()) {  
 $this->bootProvider($provider);  
 }  
  
 **return** $provider;  
}

可以看到register方法中主要的是

*//运行服务的注册方法*$provider->register();

*//将给定的服务提供者标记为已注册*$this->markAsRegistered($provider);

*//指明程序是由已启动 未启动为false***if** ($this->isBooted()) {  
 $this->bootProvider($provider);  
}

最后返回的是注册的服务，

而EventServiceProvider、LogServiceProvider、RoutingServiceProvider

的rigister()方法中都是单例的绑定像这样

**public function** register()  
{  
 $this->**app**->singleton(**'events'**, **function** ($app) {  
 **return** (**new** Dispatcher($app))->setQueueResolver(**function** () **use** ($app) {  
 **return** $app->make(QueueFactoryContract::***class***);  
 });  
 });  
}

;

##### 2.14$this->registerCoreContainerAliases();

*/将核心类的别名注册到容器中 即$alias=[]数组*

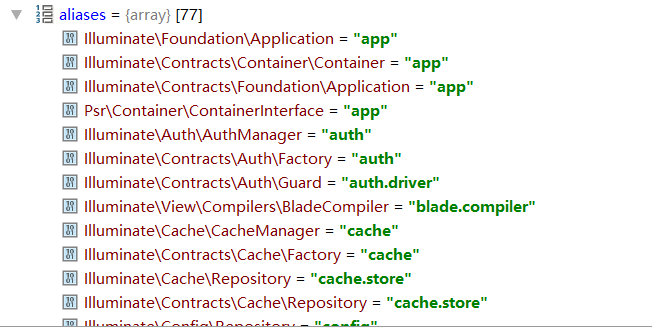
**public function** registerCoreContainerAliases()  
{  
 **foreach** ([  
 **'app'** => [**self**::***class***, \Illuminate\Contracts\Container\Container::***class***, \Illuminate\Contracts\Foundation\Application::***class***, \Psr\Container\ContainerInterface::***class***],  
 **'auth'** => [\Illuminate\Auth\AuthManager::***class***, \Illuminate\Contracts\Auth\Factory::***class***],  
 **'auth.driver'** => [\Illuminate\Contracts\Auth\Guard::***class***],  
 **'blade.compiler'** => [\Illuminate\View\Compilers\BladeCompiler::***class***],  
 **'cache'** => [\Illuminate\Cache\CacheManager::***class***, \Illuminate\Contracts\Cache\Factory::***class***],  
 **'cache.store'** => [\Illuminate\Cache\Repository::***class***, \Illuminate\Contracts\Cache\Repository::***class***],  
 **'config'** => [\Illuminate\Config\Repository::***class***, \Illuminate\Contracts\Config\Repository::***class***],  
 **'cookie'** => [\Illuminate\Cookie\CookieJar::***class***, \Illuminate\Contracts\Cookie\Factory::***class***, \Illuminate\Contracts\Cookie\QueueingFactory::***class***],  
 **'encrypter'** => [\Illuminate\Encryption\Encrypter::***class***, \Illuminate\Contracts\Encryption\Encrypter::***class***],  
 **'db'** => [\Illuminate\Database\DatabaseManager::***class***],  
 **'db.connection'** => [\Illuminate\Database\Connection::***class***, \Illuminate\Database\ConnectionInterface::***class***],  
 **'events'** => [\Illuminate\Events\Dispatcher::***class***, \Illuminate\Contracts\Events\Dispatcher::***class***],  
 **'files'** => [\Illuminate\Filesystem\Filesystem::***class***],  
 **'filesystem'** => [\Illuminate\Filesystem\FilesystemManager::***class***, \Illuminate\Contracts\Filesystem\Factory::***class***],  
 **'filesystem.disk'** => [\Illuminate\Contracts\Filesystem\Filesystem::***class***],  
 **'filesystem.cloud'** => [\Illuminate\Contracts\Filesystem\Cloud::***class***],  
 **'hash'** => [\Illuminate\Hashing\HashManager::***class***],  
 **'hash.driver'** => [\Illuminate\Contracts\Hashing\Hasher::***class***],  
 **'translator'** => [\Illuminate\Translation\Translator::***class***, \Illuminate\Contracts\Translation\Translator::***class***],  
 **'log'** => [\Illuminate\Log\LogManager::***class***, \Psr\Log\LoggerInterface::***class***],  
 **'mailer'** => [\Illuminate\Mail\Mailer::***class***, \Illuminate\Contracts\Mail\Mailer::***class***, \Illuminate\Contracts\Mail\MailQueue::***class***],  
 **'auth.password'** => [\Illuminate\Auth\Passwords\PasswordBrokerManager::***class***, \Illuminate\Contracts\Auth\PasswordBrokerFactory::***class***],  
 **'auth.password.broker'** => [\Illuminate\Auth\Passwords\PasswordBroker::***class***, \Illuminate\Contracts\Auth\PasswordBroker::***class***],  
 **'queue'** => [\Illuminate\Queue\QueueManager::***class***, \Illuminate\Contracts\Queue\Factory::***class***, \Illuminate\Contracts\Queue\Monitor::***class***],  
 **'queue.connection'** => [\Illuminate\Contracts\Queue\Queue::***class***],  
 **'queue.failer'** => [\Illuminate\Queue\Failed\FailedJobProviderInterface::***class***],  
 **'redirect'** => [\Illuminate\Routing\Redirector::***class***],  
 **'redis'** => [\Illuminate\Redis\RedisManager::***class***, \Illuminate\Contracts\Redis\Factory::***class***],  
 **'request'** => [\Illuminate\Http\Request::***class***, \Symfony\Component\HttpFoundation\Request::***class***],  
 **'router'** => [\Illuminate\Routing\Router::***class***, \Illuminate\Contracts\Routing\Registrar::***class***, \Illuminate\Contracts\Routing\BindingRegistrar::***class***],  
 **'session'** => [\Illuminate\Session\SessionManager::***class***],  
 **'session.store'** => [\Illuminate\Session\Store::***class***, \Illuminate\Contracts\Session\Session::***class***],  
 **'url'** => [\Illuminate\Routing\UrlGenerator::***class***, \Illuminate\Contracts\Routing\UrlGenerator::***class***],  
 **'validator'** => [\Illuminate\Validation\Factory::***class***, \Illuminate\Contracts\Validation\Factory::***class***],  
 **'view'** => [\Illuminate\View\Factory::***class***, \Illuminate\Contracts\View\Factory::***class***],  
 ] **as** $key => $aliases) {  
 **foreach** ($aliases **as** $alias) {  
 $this->alias($key, $alias);  
 }  
 }  
}

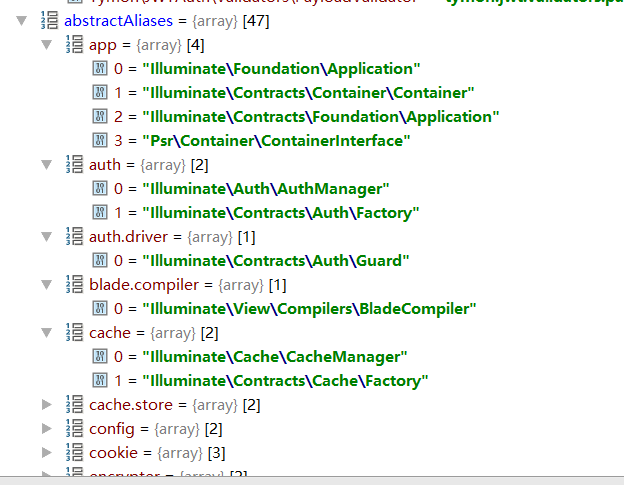
**aliases abstractAliases数组存储形式**

*他们都是已注册的类型别名不同是键值 不一样 是***aliases一维***数组*

**abstractAliases是二维数组**

**public function** alias($abstract, $alias)  
{  
 **if** ($alias === $abstract) {  
 **throw new** LogicException(**"[**{$abstract}**] is aliased to itself."**);  
 }  
 *//value==>key* $this->**aliases**[$alias] = $abstract;  
 *//key[]=value* $this->**abstractAliases**[$abstract][] = $alias;  
}





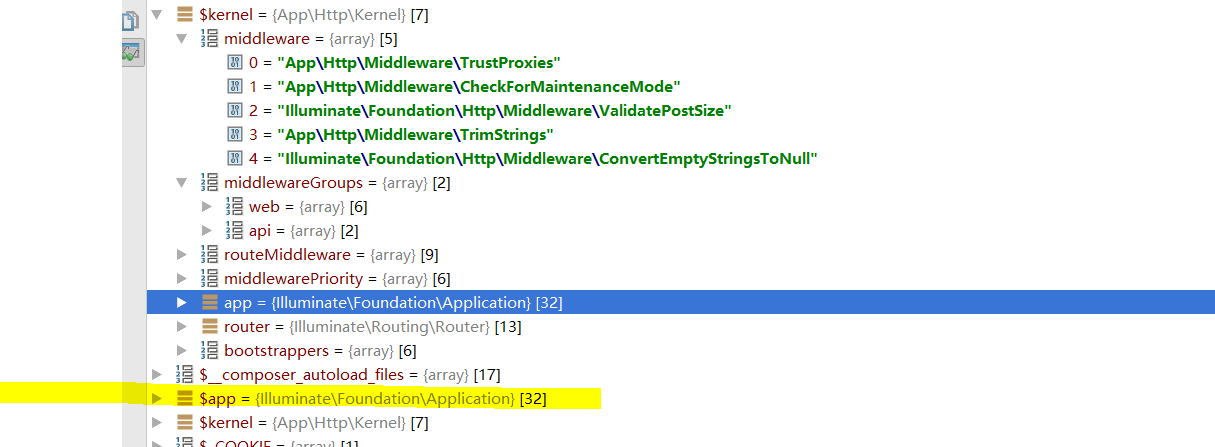
#### 2.2注册三个单例到ioc容器中

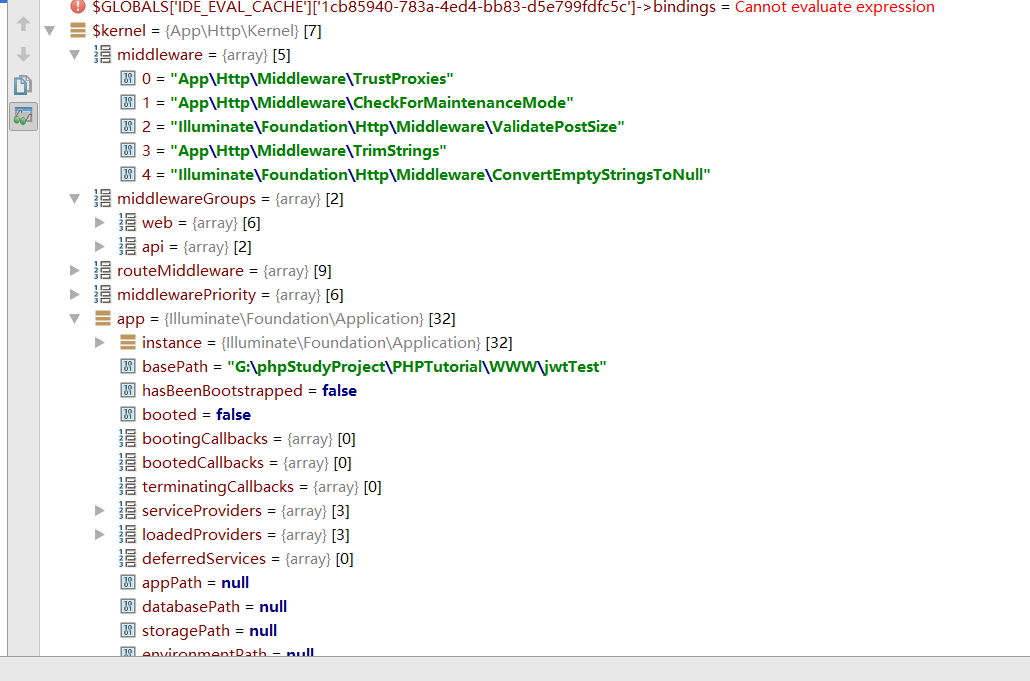
*分别是 http处理核心类（App\Http\Kernel）,cli处理核心类（App\Console\Kernel）,和异常处理（App\Exceptions\Handler）.*

$app->singleton(  
 Illuminate\Contracts\Http\Kernel::***class***,  
 App\Http\Kernel::***class***);  
  
$app->singleton(  
 Illuminate\Contracts\Console\Kernel::***class***,  
 App\Console\Kernel::***class***);  
  
$app->singleton(  
 Illuminate\Contracts\Debug\ExceptionHandler::***class***,  
 App\Exceptions\Handler::***class***);

### 3.$kernel=$app->make(Illuminate\Contracts\Http\Kernel::*class*);

$kernel数组的形式





*//获取App\Http\Kernel 实例*

主要是make()方法从容器中解析实例 即传入参数为 Illuminate\Contracts\Http\Kernel::***class***

**public function** make($abstract, **array** $parameters = [])  
{  
 *//获取别名 因为*

***//一开始初始化的别名数组alias并没有//***Illuminate\Contracts\Http\Kernel::***class 所以***abstract= abstract$abstract = $this->getAlias($abstract);   
  
 **if** ($this->isDeferredService($abstract) && ! **isset**($this->**instances**[$abstract])) {  
 $this->loadDeferredProvider($abstract);  
 }  
  
 **return parent**::*make*($abstract, $parameters);  
}

**public function** make($abstract, **array** $parameters = [])  
{  
 **return** $this->resolve($abstract, $parameters);  
}

**protected function** resolve($abstract, $parameters = [], $raiseEvents = **true**)  
{  
 *//递归获取抽象类名称的别名--为什么要递归？* $abstract = $this->getAlias($abstract);  
 // parameters为空不走 needsContextualBuild=false  
 $needsContextualBuild = ! **empty**($parameters) || ! *is\_null*(  
 *//获取上下文的具体绑定 有疑问* $this->getContextualConcrete($abstract)  
 );  
  
 *//* needsContextualBuild*为true不走  
 //如果该类型的实例当前作为单例进行管理，  
 //我们将只返回一个现有实例，而不是实例化新实例，这样开发人员就可以每次都使用相同的对象实例。*

*//* **instances数组不存在instances**[$*Illuminate\Contracts\Http\Kernel::class*]**if** (**isset**($this->**instances**[$abstract]) && ! $needsContextualBuild) {  
 **return** $this->**instances**[$abstract];  
 }  
 *//$with 参数重写堆栈数组* $this->**with**[] = $parameters;  
 *//根据别名获取指定实例 Illuminate\Contracts\Http\Kernel::class没有注册过所以$concrete=$abstract  
 //返回的 $concrete=App\Http\Kernel  
 //index.php $app->make(Illuminate\Contracts\Http\Kernel::class);  
 //在app.php Illuminate\Contracts\Http\Kernel::class, App\Http\Kernel::class已经单例绑定过*

*//第一次传递Illuminate\Contracts\Http\Kernel::class $concrete是一个闭包  
//因为第一次singleton绑定的时候出发bind方法在bindings数组中存放的是闭包*

$concrete = $this->getConcrete($abstract);  
  
 *//我们准备实例化为绑定注册的具体类型的实例。  
 //这将实例化类型，并递归地解析其任何“嵌套”依赖项，直到所有的依赖项都得到解决。  
 //return $concrete === $abstract || $concrete instanceof Closure;* **if** ($this->isBuildable($concrete, $abstract)) {  
 *//实例化给定类型的具体实例 build利用反射创建实例* $object = $this->build($concrete);  
 } **else** {  
 $object = $this->make($concrete);  
 }  
**foreach** ($this->getExtenders($abstract) **as** $extender) {  
 $object = $extender($object, $this);  
 }  
  
**if** ($this->isShared($abstract) && ! $needsContextualBuild) {  
 $this->**instances**[$abstract] = $object;  
 }  
  
 **if** ($raiseEvents) {  
 $this->fireResolvingCallbacks($abstract, $object);  
 }  
  
$this->**resolved**[$abstract] = **true**;  
  
 *array\_pop*($this->**with**);  
  
 **return** $object;  
}

---

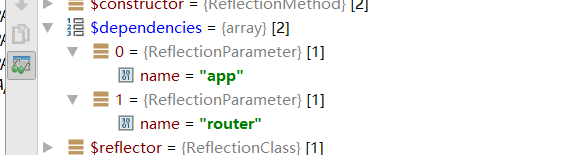
**public function** build($concrete)  
{  
 *// If the concrete type is actually a Closure, we will just execute it and  
 // hand back the results of the functions, which allows functions to be  
 // used as resolvers for more fine-tuned resolution of these objects.  
 //如果$concrete是闭包函数则执行这个函数*

*//传入的是$this->bindings[Illuminate\Contracts\Http\Kernel::class]的闭包* **if** ($concrete **instanceof** Closure) {  
 **return** $concrete($this, $this->getLastParameterOverride());  
 }

//第二次执行resolve是拿到app\http\kernel类 执行反射 进行一些初始化操作  
 *//不是匿名函数 反射 把具体的类放入堆栈数组* $reflector = **new** ReflectionClass($concrete);  
  
 *// If the type is not instantiable, the developer is attempting to resolve  
 // an abstract type such as an Interface or Abstract Class and there is  
 // no binding registered for the abstractions so we need to bail out.* **if** (! $reflector->isInstantiable()) {  
 **return** $this->notInstantiable($concrete);  
 }  
  
 $this->**buildStack**[] = $concrete;  
  
 $constructor = $reflector->getConstructor();  
  
 *// If there are no constructors, that means there are no dependencies then  
 // we can just resolve the instances of the objects right away, without  
 // resolving any other types or dependencies out of these containers.* **if** (*is\_null*($constructor)) {  
 *array\_pop*($this->**buildStack**);  
  
 **return new** $concrete;  
 }  
  
 $dependencies = $constructor->getParameters();

//反射为啥会获取到app router疑惑

//dependencies格式

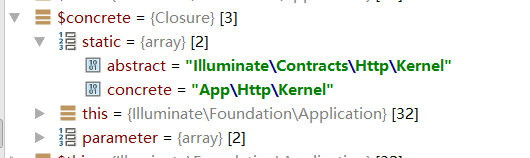
  
  
 *// Once we have all the constructor's parameters we can create each of the  
 // dependency instances and then use the reflection instances to make a  
 // new instance of this class, injecting the created dependencies in.* **try** {

//*//会根据app\http\kernel下的数组不断make*

$instances = $this->resolveDependencies($dependencies);  
 } **catch** (BindingResolutionException $e) {  
 *array\_pop*($this->**buildStack**);  
  
 **throw** $e;  
 }  
  
 *array\_pop*($this->**buildStack**);  
  
 **return** $reflector->newInstanceArgs($instances);  
}

--执行闭包函数 再次执行resolve（）方法传递参数如下





**protected function** getClosure($abstract, $concrete)  
{  
 **return function** ($container, $parameters = []) **use** ($abstract, $concrete) {  
 **if** ($abstract == $concrete) {  
 **return** $container->build($concrete);  
 }  
  
 **return** $container->resolve(  
 $concrete, $parameters, $raiseEvents = **false** );  
 };  
}

*/\*\*  
 \* Get the alias for an abstract if available.  
 \*获取抽象类名称的别名  
 \** ***@param*** *string $abstract  
 \** ***@return*** *string  
 \*/***public function** getAlias($abstract)  
{  
 **if** (! **isset**($this->**aliases**[$abstract])) {  
 **return** $abstract;  
 }  
  
 **return** $this->getAlias($this->**aliases**[$abstract]);  
}

最主要的是resolve（）方法

### 4. $kernel->handle

注册http处理核心类（App\Http\Kernel）、cli处理核心类（App\Console\Kernel）和异常处理（App\Exceptions\Handler）

注册中间件

完成http请求环境的初始化

运行中间件

源码：

Handle()的主要作用是处理传入的http请求

**public function** handle($request)  
{  
 **try** {  
 $request->enableHttpMethodParameterOverride();  
  
 *//通过中间件或路由发送给定的请求* $response = $this->sendRequestThroughRouter($request);  
 } **catch** (Exception $e) {  
 $this->reportException($e);  
  
 $response = $this->renderException($request, $e);  
 } **catch** (Throwable $e) {  
 $this->reportException($e = **new** FatalThrowableError($e));  
  
 $response = $this->renderException($request, $e);  
 }  
 *//运行事件对应的监听器类 --dispatch有疑惑* $this->**app**[**'events'**]->dispatch(  
 **new** Events\RequestHandled($request, $response)  
 );  
  
 **return** $response;  
}

*/\*\*  
 \* Send the given request through the middleware / router.  
 \* 通过中间件或路由发送给定的请求  
 \** ***@param*** *\Illuminate\Http\Request $request  
 \** ***@return*** *\Illuminate\Http\Response  
 \*/***protected function** sendRequestThroughRouter($request)  
{  
 $this->**app**->instance(**'request'**, $request);  
 Facade::*clearResolvedInstance*(**'request'**);  
 *//Bootstrap the application for HTTP requests  
 //启动应用程序的http请求*

*//各种服务的注册与启动*

$this->bootstrap();  
 **return** (**new** Pipeline($this->**app**))//请求的分发  
 ->send($request)  
 ->through($this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**)  
 ->then($this->dispatchToRouter());  
}

--

*//启动应用程序的http请求*

**public function** bootstrap()  
{  
 **if** (! $this->**app**->hasBeenBootstrapped()) {  
 $this->**app**->bootstrapWith($this->bootstrappers());  
 }  
}

### 5.运行Illuminate\Http\Response父类的send方法响应请求

### 6.HTTP 响应被发送到浏览器之后，调用中间件的terminate()方法

# ioc容器

**服务容器主要的两个作用绑定与解析**

## 关于ioc容器的理解

参考：<https://www.insp.top/learn-laravel-container>

https://blog.csdn.net/weixin\_40909770/article/details/79125709

对于laravel 学习笔记 —— 神奇的服务容器的理解

这篇文章以超人与超能力为例

1. 直接在superman类中new 各种超能力实例
2. 在工厂中根据不用逻辑new不同超能力实例，superman类中只有工厂依赖
3. 将各种超能力用接口统一，superman持有抽象接口的依赖
4. Ioc容器—先进的工厂，通过绑定脚本，实现工厂自动化生产

class Container

{

protected $binds;

protected $instances;

public function bind($abstract, $concrete)

{

if ($concrete instanceof Closure) {

$this->binds[$abstract] = $concrete;

} else {

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

}

}

public function make($abstract, $parameters = [])

{

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

return $this->instances[$abstract];

}

array\_unshift($parameters, $this);

return call\_user\_func\_array($this->binds[$abstract], $parameters);

}

}

容器中有两个主要的方法 make和bind

Bind定义了两个数组,binds数组按key值存放了每个服务，它的值是个闭包用来延迟加载一些服务，从中我们可以看到，绑定的真正意思就是将我们所需服务的回调函数或者实例存入服务容器中，以备将来我们直接调用。

make

首先判断我们所需服务是如何绑定到服务容器的，如果是实例绑定的，则直接从instances数组中取出实例返回，如果不是的话，则使用call\_user\_func\_array函数回调我们存储在binds函数中的闭包函数。array\_unshift函数是把服务容器存储在parameter数组中，供回调函数使用

// 创建一个容器（后面称作超级工厂）

$container = new Container;

// 向该 超级工厂 添加 超人 的生产脚本

$container->bind('superman', function($container, $moduleName) {

return new Superman($container->make($moduleName));

});

// 向该 超级工厂 添加 超能力模组 的生产脚本

$container->bind('xpower', function($container) {

return new XPower;

});

// 同上

$container->bind('ultrabomb', function($container) {

return new UltraBomb;

});

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 华丽丽的分割线 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 开始启动生产

$superman\_1 = $container->make('superman', ['xpower']);

$superman\_2 = $container->make('superman', ['ultrabomb']);

$superman\_3 = $container->make('superman', ['xpower']);

执行顺序的分析：

1. new 容器类
2. 绑定超人生产脚本 由于传递的参数是闭包所以

bind[‘superman’]= function($container, $moduleName) {

return new Superman($container->make($moduleName));

})

开始生产

$instance数组不存在以superman为key的值

$this指向当前容器 array\_ unshift()将当前容器添加到['xpower']数组

变成[$container,’xpower’]

Call\_user\_func\_array($this->bind[‘superman’], [$container,’xpower’])回调

即执行

function($container, ’ xpower’’) {

return new Superman($container->make(‘’ xpower’’));

});

$superman\_1 = $container->make('superman', ['xpower']);

同理执行$container->make('superman', ['xpower']

call\_user\_func\_array($this->binds[‘'xpower'’], [$container]);

function($container) {

return new XPower;

});

返回 new XPower

总结

$superman\_1 = $container->make('superman', ['xpower']);

等价于

$superman\_1= new Superman(new XPower));

通过最初的 绑定（bind） 操作，我们向 超级工厂 注册了一些生产脚本，这些生产脚本在生产指令下达之时便会执行我们彻底的解除了 超人 与 超能力模组 的依赖关系，更重要的是，容器类也丝毫没有和他们产生任何依赖！我们通过注册、绑定的方式向容器中添加一段可以被执行的回调（可以是匿名函数、非匿名函数、类的方法）作为生产一个类的实例的 **脚本** ，只有在真正的 生产（make） 操作被调用执行时，才会触发。

## 2.laravel Ioc源码解析

参考: <https://blog.csdn.net/kwinH/article/details/56284633?utm_source=blogxgwz9>

**\Illuminate\Container\Container类是Laravel中容器的实现**

### ioc容器何时创建

### 2.ioc容器的绑定

将服务绑定到容器有4中方法 instance、bind、singleton、alias

#### Instance

Instance绑定实例，将一个已存在的对象绑定到容器中，然后通过名称解析该服务时，容器返回这个绑定的实例。

$api = new HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\Api', $api);

值得注意的是绑定时的别名是按照实例的路径起的

##### Instance源码分析：

/\*\*  
 \* Register an existing instance as shared in the container.  
 \* 将现有实例注册为容器中的共享实例  
 \* **@param** string $abstract  
 \* **@param** mixed $instance  
 \* **@return** mixed  
 \*/  
 **public function** instance($abstract, $instance)  
 {

*//从上下文绑定别名缓存中删除别名  
// 此方法完成工作：如果$this->aliases[$searched]即要绑定实例的别名不存在 return  
// 否则删除abstractAliases数组中相应的值，//unset($this->abstractAliases[$abstract][$index]);*

$this->removeAbstractAlias($abstract);  
  
 $isBound = $this->bound($abstract);  
  *//如果aliases数组存在绑定的别名则删除 不存在删除也不报错*

**unset**($this->**aliases**[$abstract]);  
  
*// We'll check to determine if this type has been bound before, and if it has  
// we will fire the rebound callbacks registered with the container and it  
// can be updated with consuming classes that have gotten resolved here.*

*//赋值* $this->**instances**[$abstract] = $instance;  
*/*

*/确定给定的抽象类型是否已绑定*

**if** ($isBound) {  
 $this->rebound($abstract);  
 }  
  
 **return** $instance;  
 }

从上面的instance方法可以看出instance($abstract, $instance)方法主要作用是将现有实例注册为容器中的共享实例。

##### Instance()执行步骤

###### $this->removeAbstractAlias($abstract);

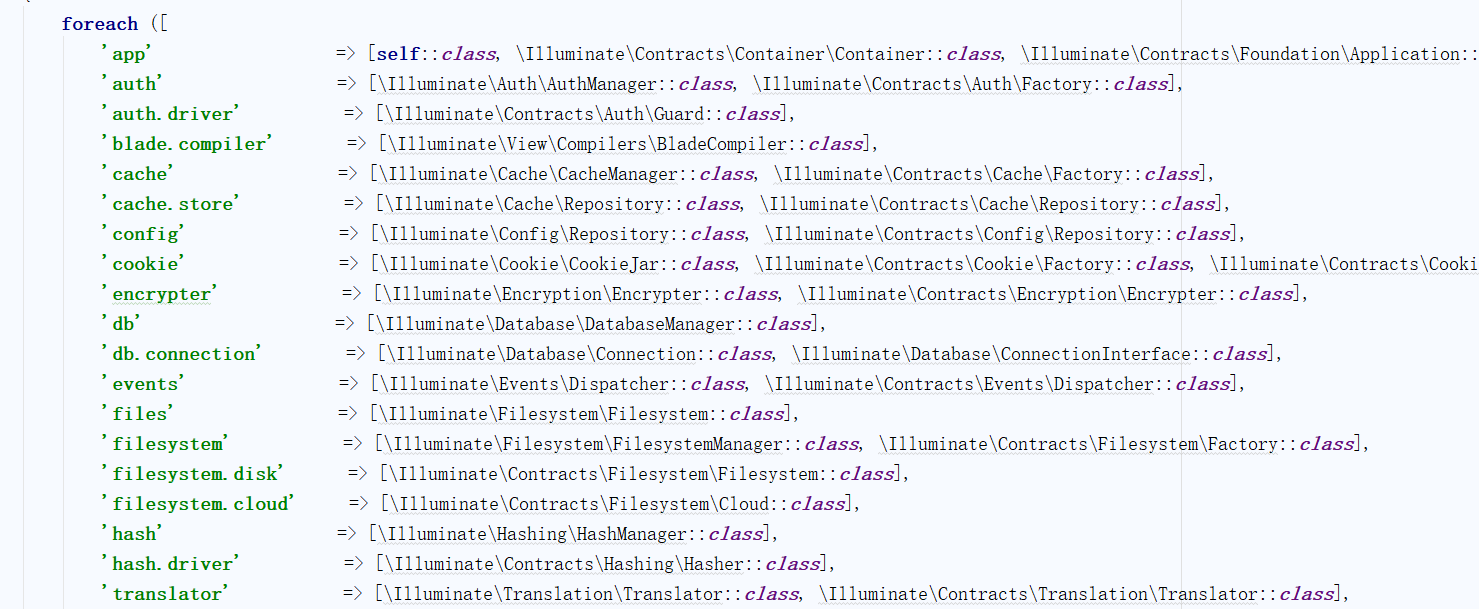
这个方法源码的解释是：从上下文绑定别名缓存中删除别名,

此方法完成工作：如果$this->aliases[$searched]即要绑定实例的别名不存在 return  
否则删除abstractAliases数组中相应的值，unset($this->abstractAliases[$abstract][$index]);

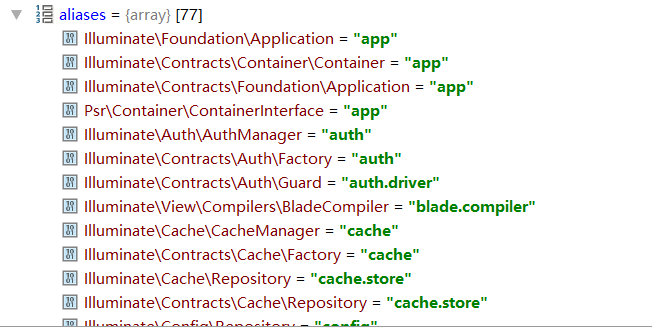
**protected function** removeAbstractAlias($searched)  
{  
 *//$aliases=[] 存放已注册的类型别名的数组* **if** (! **isset**($this->**aliases**[$searched])) {  
 **return**;  
 }  
 *//$abstractAliases=[] 以抽象名称为key的已注册别名数组* **foreach** ($this->**abstractAliases as** $abstract => $aliases) {  
 **foreach** ($aliases **as** $index => $alias) {  
 **if** ($alias == $searched) {  
 **unset**($this->**abstractAliases**[$abstract][$index]);  
 }  
 }  
 }  
}

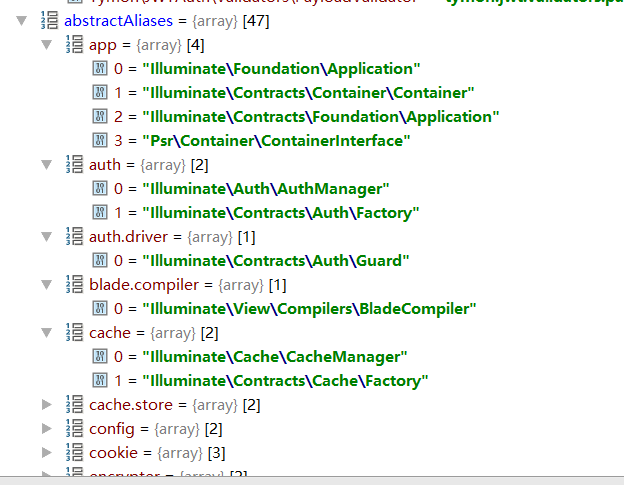
关于$aliases $abstractAliases

初始化在服务容器初始化的时候$this->registerCoreContainerAliases();完成，一开始$aliases $abstractAliases初始化的数据如下



程序启动后的数组值





###### 2. $isBound = $this->bound($abstract);

**public function** bound($abstract)  
{  
 **return isset**($this->**bindings**[$abstract]) ||  
 **isset**($this->**instances**[$abstract]) ||  
 $this->isAlias($abstract);  
}

返回 true or false

###### 3. unset($this->aliases[$abstract]);

*//unset() 删除不存在变量 不报错*

如果aliases数组存在绑定的别名则删除 不存在删除也不报错

###### 4．$this->instances[$abstract] = $instance;

要绑定的别名不存在instance实例数组赋值

###### if ($isBound)

**if** ($isBound) {  
 $this->rebound($abstract);  
}

//具体执行的哪些回调 以及这些回调函数的初始化不是很理解

**protected function** rebound($abstract)  
{  
 *//获取绑定的实例—make源码的具体放在后面* $instance = $this->make($abstract);  
 //执行存放 别名回调函数的数组中的所有回调函数  
 **foreach** ($this->getReboundCallbacks($abstract) **as** $callback) {  
 *call\_user\_func*($callback, $this, $instance);  
 }  
}

**protected function** getReboundCallbacks($abstract)  
{  
 *//$reboundCallbacks=[]存放已注册的rebound回调函数数组* **return** $this->**reboundCallbacks**[$abstract] ?? [];  
}

#### bind绑定

##### 绑定自身

$this->app->bind('App\Services\RedisEventPusher', null);

##### 绑定闭包

$this->app->bind('HelpSpot\API', function ($app) {

return new HelpSpot\API();

});//闭包直接提供实现方式

$this->app->bind('HelpSpot\API', function ($app) {

return new HelpSpot\API($app->make('HttpClient'));

});//需要依赖注入

##### 绑定接口

$this->app->bind( 'App\Contracts\EventPusher', 'App\Services\RedisEventPusher' );

**源码：**

**public function** bind($abstract, $concrete = **null**, $shared = **false**)  
{  
 *// unset($this->instances[$abstract], $this->aliases[$abstract]);  
 //删除 之前以$abstract别名的实例 和 别名数组* $this->dropStaleInstances($abstract);  
 *//如果没有指定具体类型，我们只需将具体类型设置为抽象类型。  
 //之后，将注册为共享的具体类型而不必在两个参数中声明它们的类* **if** (*is\_null*($concrete)) {  
 $concrete = $abstract;  
 }  
  
 *//如果工厂不是闭包，这意味着它只是一个类名，它被绑定到这个抽象类型的容器中，  
 //们将把它包装在它自己的闭包中，以便扩展时更方便  
 // up inside its own Closure to give us more convenience when extending.* **if** (! $concrete **instanceof** Closure) {  
 $concrete = $this->getClosure($abstract, $concrete);  
 }  
 *//创建一个包含$concrete $shared变量名和它们的数值的数组* $this->**bindings**[$abstract] = *compact*(**'concrete'**, **'shared'**);  
 *//如果抽象类型已经在这个容器中解析了，  
 //我们将激活反弹侦听器，这样任何已经解析的对象都可以通过侦听器回调更新它们的对象副本* **if** ($this->resolved($abstract)) {  
 $this->rebound($abstract);  
 }  
}

##### bind源码分析

###### 1. $this->dropStaleInstances($abstract);

*//删除 之前以$abstract别名的实例 和 别名数组*

**protected function** dropStaleInstances($abstract)  
{  
 **unset**($this->**instances**[$abstract], $this->**aliases**[$abstract]);  
}

###### 2. if (is\_null($concrete)) { $concrete = $abstract; }

*//如果没有指定具体类型，我们只需将具体类型设置为抽象类型。  
//之后，将注册为共享的具体类型而不必在两个参数中声明它们的类*

###### 3. ! $concrete instanceof Closure

*//如果工厂不是闭包，这意味着它只是一个类名，它被绑定到这个抽象类型的容器中，  
//们将把它包装在它自己的闭包中，以便扩展时更方便*

**if** (! $concrete **instanceof** Closure) {

//返回闭包的方法有很多操作 暂时不明白  
 $concrete = $this->getClosure($abstract, $concrete);  
}

//*获取要在构建类型时使用的闭包*

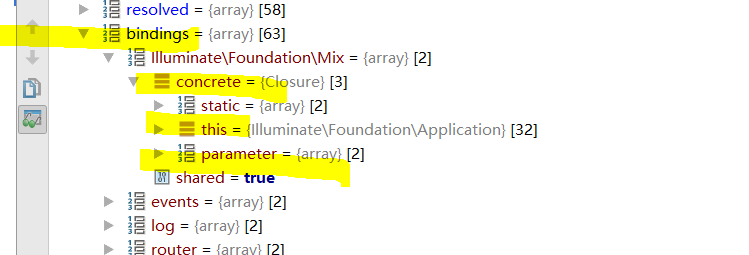
**protected function** getClosure($abstract, $concrete)  
{  
 *//返回一个闭包* **return function** ($container, $parameters = []) **use** ($abstract, $concrete) {  
 **if** ($abstract == $concrete) {  
 **return** $container->build($concrete);  
 }  
 **return** $container->resolve(  
 $concrete, $parameters, $raiseEvents = **false** );  
 };  
}

*/\*\*  
 \* Instantiate a concrete instance of the given type.  
 \*实例化给定类型的具体实例  
 \** ***@param*** *string $concrete  
 \** ***@return*** *mixed  
 \*  
 \** ***@throws*** *\Illuminate\Contracts\Container\BindingResolutionException  
 \*/***public function** build($concrete)  
{  
 *// If the concrete type is actually a Closure, we will just execute it and  
 // hand back the results of the functions, which allows functions to be  
 // used as resolvers for more fine-tuned resolution of these objects.* **if** ($concrete **instanceof** Closure) {  
 **return** $concrete($this, $this->getLastParameterOverride());  
 }  
  
 $reflector = **new** ReflectionClass($concrete);  
  
 *// If the type is not instantiable, the developer is attempting to resolve  
 // an abstract type such as an Interface or Abstract Class and there is  
 // no binding registered for the abstractions so we need to bail out.* **if** (! $reflector->isInstantiable()) {  
 **return** $this->notInstantiable($concrete);  
 }  
  
 $this->**buildStack**[] = $concrete;  
  
 $constructor = $reflector->getConstructor();  
  
 *// If there are no constructors, that means there are no dependencies then  
 // we can just resolve the instances of the objects right away, without  
 // resolving any other types or dependencies out of these containers.* **if** (*is\_null*($constructor)) {  
 *array\_pop*($this->**buildStack**);  
  
 **return new** $concrete;  
 }  
  
 $dependencies = $constructor->getParameters();  
  
 *// Once we have all the constructor's parameters we can create each of the  
 // dependency instances and then use the reflection instances to make a  
 // new instance of this class, injecting the created dependencies in.* **try** {  
 $instances = $this->resolveDependencies($dependencies);  
 } **catch** (BindingResolutionException $e) {  
 *array\_pop*($this->**buildStack**);  
  
 **throw** $e;  
 }  
  
 *array\_pop*($this->**buildStack**);  
  
 **return** $reflector->newInstanceArgs($instances);  
}

*/\*\*  
 \* Resolve the given type from the container.  
 \*从容器中解析给定类型  
 \** ***@param*** *string $abstract  
 \** ***@param*** *array $parameters  
 \** ***@param*** *bool $raiseEvents  
 \** ***@return*** *mixed  
 \*  
 \** ***@throws*** *\Illuminate\Contracts\Container\BindingResolutionException  
 \*/***protected function** resolve($abstract, $parameters = [], $raiseEvents = **true**)  
{  
 $abstract = $this->getAlias($abstract);  
  
 $needsContextualBuild = ! **empty**($parameters) || ! *is\_null*(  
 $this->getContextualConcrete($abstract)  
 );  
  
 *// If an instance of the type is currently being managed as a singleton we'll  
 // just return an existing instance instead of instantiating new instances  
 // so the developer can keep using the same objects instance every time.* **if** (**isset**($this->**instances**[$abstract]) && ! $needsContextualBuild) {  
 **return** $this->**instances**[$abstract];  
 }  
  
 $this->**with**[] = $parameters;  
  
 $concrete = $this->getConcrete($abstract);  
  
 *// We're ready to instantiate an instance of the concrete type registered for  
 // the binding. This will instantiate the types, as well as resolve any of  
 // its "nested" dependencies recursively until all have gotten resolved.* **if** ($this->isBuildable($concrete, $abstract)) {  
 $object = $this->build($concrete);  
 } **else** {  
 $object = $this->make($concrete);  
 }  
  
 *// If we defined any extenders for this type, we'll need to spin through them  
 // and apply them to the object being built. This allows for the extension  
 // of services, such as changing configuration or decorating the object.* **foreach** ($this->getExtenders($abstract) **as** $extender) {  
 $object = $extender($object, $this);  
 }  
  
 *// If the requested type is registered as a singleton we'll want to cache off  
 // the instances in "memory" so we can return it later without creating an  
 // entirely new instance of an object on each subsequent request for it.* **if** ($this->isShared($abstract) && ! $needsContextualBuild) {  
 $this->**instances**[$abstract] = $object;  
 }  
  
 **if** ($raiseEvents) {  
 $this->fireResolvingCallbacks($abstract, $object);  
 }  
  
 *// Before returning, we will also set the resolved flag to "true" and pop off  
 // the parameter overrides for this build. After those two things are done  
 // we will be ready to return back the fully constructed class instance.* $this->**resolved**[$abstract] = **true**;  
  
 *array\_pop*($this->**with**);  
  
 **return** $object;  
}

###### 4. $this->bindings[$abstract] = *compact*('concrete', 'shared');

###### 5. if ($this->resolved($abstract)) { $this->rebound($abstract); }



##### 疑问

*//删除所有过时的实例和别名*$this->dropStaleInstances($abstract);

$concrete = $this->getClosure($abstract, $concrete);生成闭包的理解

//为什么要判断他是否解析过

$this->resolved($abstract)

### ioc容器服务解析make

#### make源码

*/\*\*  
 \* Resolve the given type from the container.  
 \* 解析服务  
 \** ***@param*** *string $abstract  
 \** ***@param*** *array $parameters  
 \** ***@return*** *mixed  
 \*  
 \** ***@throws*** *\Illuminate\Contracts\Container\BindingResolutionException  
 \*/***public function** make($abstract, **array** $parameters = [])  
{  
 **return** $this->resolve($abstract, $parameters);  
}

#### 疑惑

*//递归获取抽象类名称的别名--为什么要递归？*$abstract = $this->getAlias($abstract);

*//获取相关的具体绑定 有疑问*$this->getContextualConcrete($abstract)

# 路由

## 1.路由初始化

源码分析：  
在laravel的启动流程中有说 初始化container时对路由初始化了

Application类中

$this-> registerBaseServiceProviders ();

**protected function** registerBaseServiceProviders()  
{  
 $this->register(**new** EventServiceProvider($this));  
 $this->register(**new** LogServiceProvider($this));  
 $this->register(**new** RoutingServiceProvider($this));  
}

**RoutingServiceProvider源码：**

**class** RoutingServiceProvider **extends** ServiceProvider  
{  
 */\*\*  
 \* Register the service provider.  
 \*  
 \** ***@return*** *void  
 \*/* **public function** register()  
 {  
 $this->registerRouter();  
 $this->registerUrlGenerator();  
 $this->registerRedirector();  
 $this->registerPsrRequest();  
 $this->registerPsrResponse();  
 $this->registerResponseFactory();  
 $this->registerControllerDispatcher();  
 }

…

## route服务的启动与加载 laravel在初始化Application后 进行下面两个操作

*/\*\*  
 \* Handle an incoming HTTP request.  
 \* 处理传入的http请求  
 \** ***@param*** *\Illuminate\Http\Request $request  
 \** ***@return*** *\Illuminate\Http\Response  
 \*/***public function** handle($request)  
{  
 **try** {  
 $request->enableHttpMethodParameterOverride();  
  
 *//将请求发送至中间件、路由处理* $response = $this->sendRequestThroughRouter($request);  
 } **catch** (Exception $e) {  
 $this->reportException($e);  
  
 $response = $this->renderException($request, $e);  
 } **catch** (Throwable $e) {  
 $this->reportException($e = **new** FatalThrowableError($e));  
  
 $response = $this->renderException($request, $e);  
 }  
 *//运行事件对应的监听器类 --dispatch有疑惑* $this->**app**[**'events'**]->dispatch(  
 **new** Events\RequestHandled($request, $response)  
 );  
  
 **return** $response;  
}

*/\*\*  
 \* Send the given request through the middleware / router.  
 \* 通过中间件或路由发送给定的请求  
 \** ***@param*** *\Illuminate\Http\Request $request  
 \** ***@return*** *\Illuminate\Http\Response  
 \*/***protected function** sendRequestThroughRouter($request)  
{  
 $this->**app**->instance(**'request'**, $request);  
 Facade::*clearResolvedInstance*(**'request'**);  
 *//Bootstrap the application for HTTP requests  
 //启动应用程序的http请求  
 //各种服务的注册与启动* $this->bootstrap();  
 **return** (**new** Pipeline($this->**app**))*//请求的分发* ->send($request)  
 ->through($this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**)  
 ->then($this->dispatchToRouter());  
}

$kernel->handle调用RouteServiceProvider的register和boot两个方法

*//获取App\Http\Kernel 实例*$kernel = $app->make(Illuminate\Contracts\Http\Kernel::***class***);  
  
*/\*\*  
 \* 注册中间件  
完成http请求环境的初始化  
运行中间件  
处理传入的http请求  
 \*/*$response = $kernel->handle(  
 $request = Illuminate\Http\Request::*capture*()  
);

然后前面提到的handle方法中

*/将请求发送至中间件、路由处理*$response = $this->sendRequestThroughRouter($request);

会调用$this->bootstrap();*//各种服务的注册与启动*

这个方法的主要作用就是

调用 RouteServiceProvider的register和boot两个方法

App\Providers\ RouteServiceProvider源码

它继承了Illuminate\Foundation\Support\Providers

**class** RouteServiceProvider **extends** ServiceProvider  
{  
 */\*\*  
 \* This namespace is applied to your controller routes.  
 \*  
 \* In addition, it is set as the URL generator's root namespace.  
 \*  
 \** ***@var*** *string  
 \*/* **protected $namespace** = **'App\Http\Controllers'**;  
  
 */\*\*  
 \* Define your route model bindings, pattern filters, etc.  
 \*  
 \** ***@return*** *void  
 \*/* **public function** boot()  
 {  
 *//* **parent**::*boot*();  
 }

Illuminate\Foundation\Support\Providers\RouteServiceProvider

源码

**public function** boot()  
{  
 *//设置应用程序的根控制器命名空间* $this->setRootControllerNamespace();  
  
 *//判断是否存在缓存文件* **if** ($this->routesAreCached()) {  
 *//加载应用程序的缓存路由* $this->loadCachedRoutes();  
 } **else** {  
 *//加载应用路由* $this->loadRoutes();  
 //保存在Application->bootedCallbacks[]数组里  
 $this->**app**->booted(**function** () {  
 $this->**app**[**'router'**]->getRoutes()->refreshNameLookups();  
 $this->**app**[**'router'**]->getRoutes()->refreshActionLookups();  
 });  
 }  
}

### 2.1$this->setRootControllerNamespace();

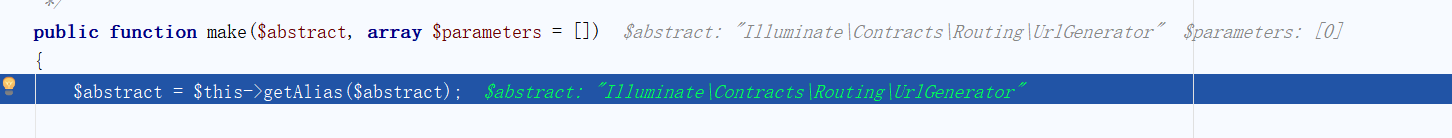
*//设置应用程序的根控制器命名空间*

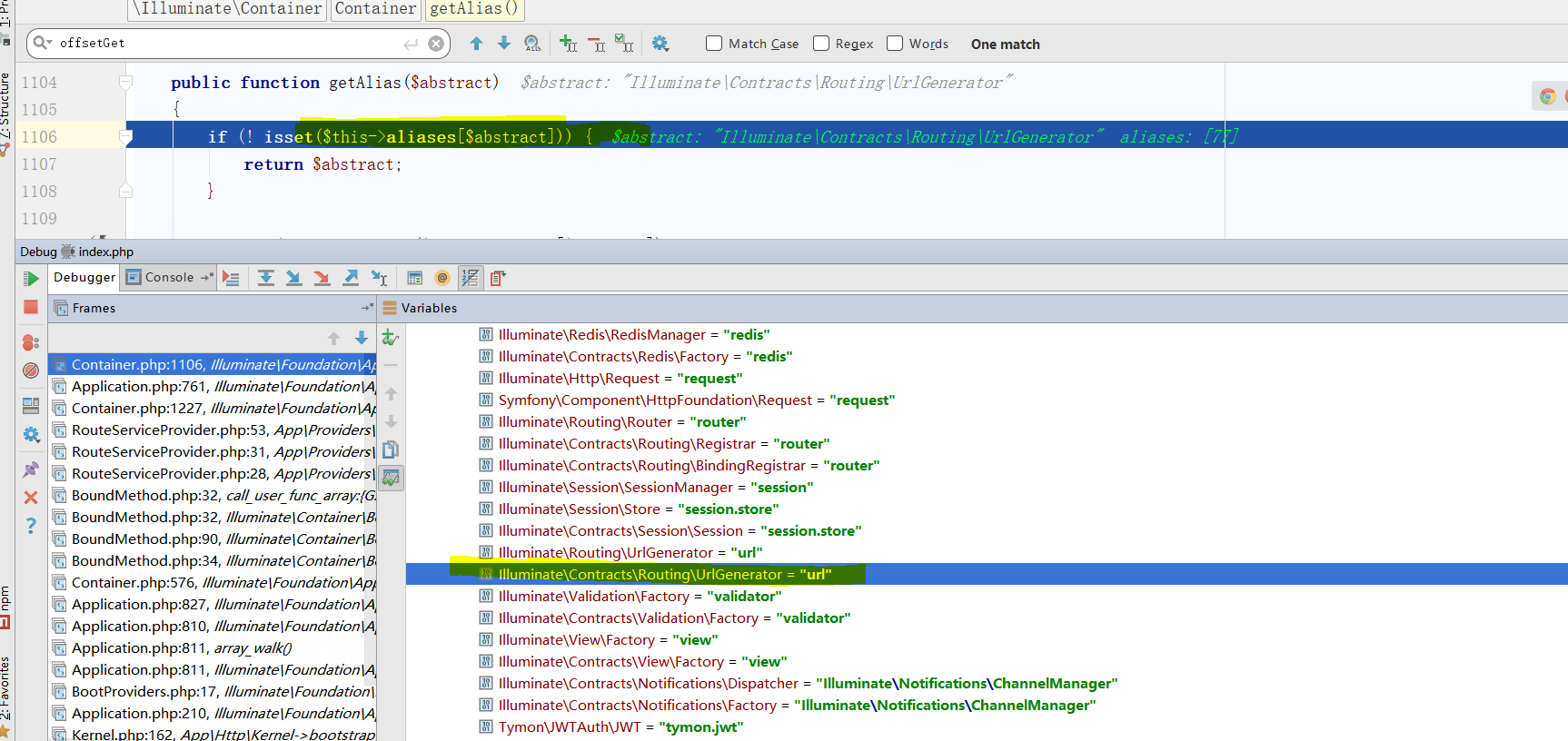
*/\*\*  
 \* Set the root controller namespace for the application.  
 \* 设置应用程序的根控制器命名空间  
 \** ***@return*** *void  
 \*/***protected function** setRootControllerNamespace()  
{  
 *// protected $namespace = 'App\Http\Controllers';* **if** (! *is\_null*($this->**namespace**)) {  
 *//$this->app[UrlGenerator::class]存放的是Illuminate\Routing\UrlGenerator 一个实例  
 //因为Illuminate\Foundation\Application 继承了Illuminate\Container\Container;  
 //而Illuminate\Container\Container;implements ArrayAccess  
 // 所以 $this->app[UrlGenerator::class]这种方式调用会触发offsetGet()* $this->**app**[UrlGenerator::***class***]->setRootControllerNamespace($this->**namespace**);  
 }  
}

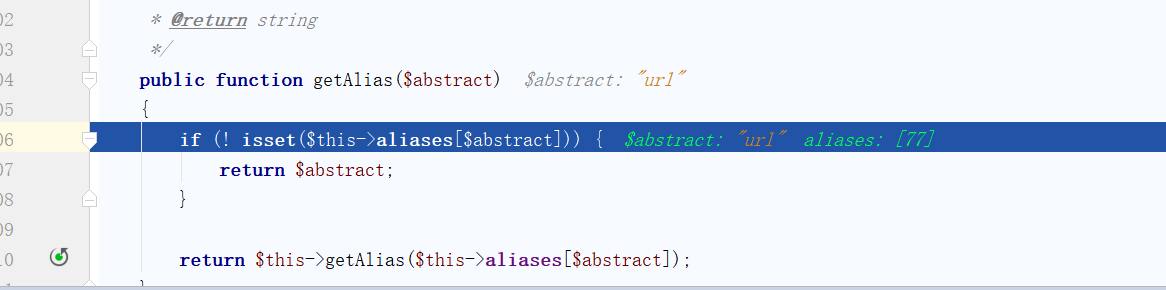
-------

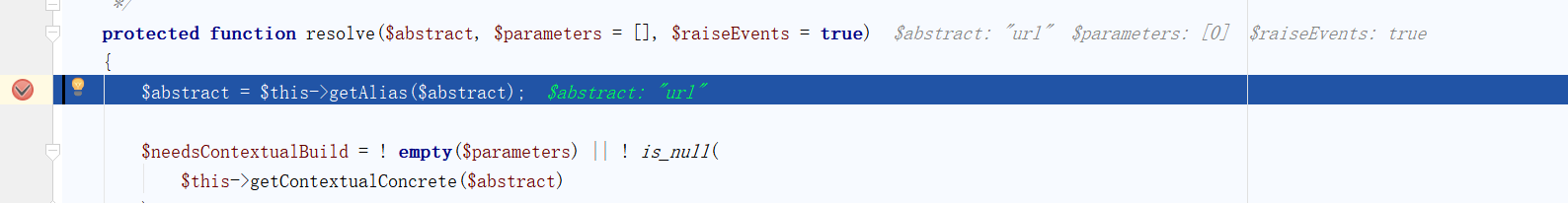
**public function** offsetGet($key)  
{  
 **return** $this->make($key);  
}

#### make(Illuminate\container\Routing\UrlGenerato)执行过程





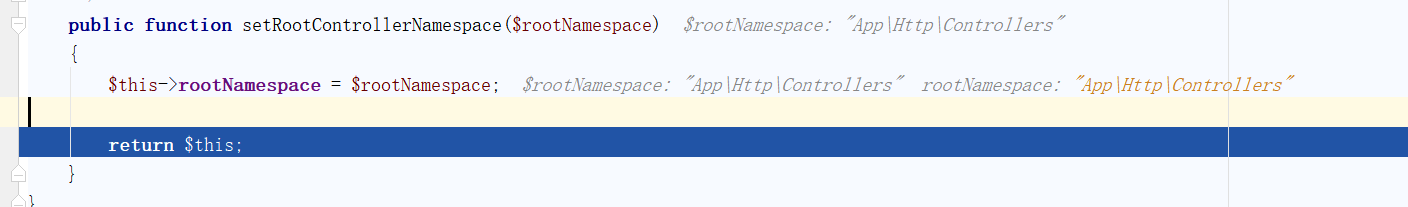


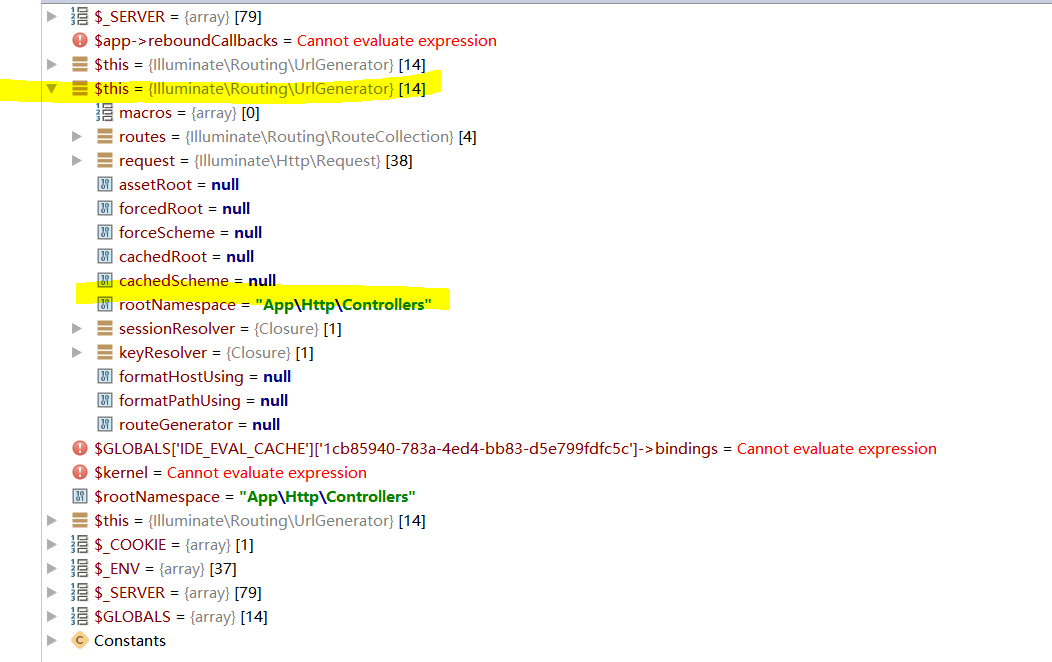


//在执行make的时候因为存放的是闭包所以build会执行闭包

-- *Illuminate\Routing\UrlGenerator 的*setRootControllerNamespace返回$this即返回他自己一个实例

最终返回





**public function** setRootControllerNamespace($rootNamespace)  
{  
 $this->**rootNamespace** = $rootNamespace;  
  
 **return** $this;  
}

### 2.2 $this->loadRoutes();*//加载应用路由*

**protected function** loadRoutes()  
{  
 **if** (*method\_exists*($this, **'map'**)) {  
 $this->**app**->call([$this, **'map'**]);  
 }  
}

--Illuminate\Container下的Container.php

**public function** call($callback, **array** $parameters = [], $defaultMethod = **null**)  
{  
 **return** BoundMethod::*call*($this, $callback, $parameters, $defaultMethod);  
}

加载路由的时候会触发这个函数，$this->**app**->call([$this, **'map'**]);记过一些列操作(这个操作有点复杂，以后再看？)调用了 app\Providers\RouteServiceProvider map()方法

**public function** map()  
{  
 $this->mapApiRoutes();  
  
 $this->mapWebRoutes();  
  
 *//*}

*//为程序定义web组 路由*

**protected function** mapWebRoutes()  
{  
 Route::*middleware*(**'web'**)  
 ->**namespace**($this->**namespace**)  
 ->group(base\_path(**'routes/web.php'**));  
}

//*为程序定义api组 路由*

**protected function** mapApiRoutes()  
{  
 Route::*prefix*(**'api'**)  
 ->**middleware**(**'api'**)  
 ->**namespace**($this->**namespace**)  
 ->group(base\_path(**'routes/api.php'**));  
}

--- mapWebRoutes方法具体如何实现？ ---过程很复杂，以后再看？主要就是加载api.php和web.php这两个文件

Illuminate\Foundation AliasLoader.php

*/\*\*  
 \* Load a class alias if it is registered.  
 \*如果类别名已注册，则加载它  
 \** ***@param*** *string $alias  
 \** ***@return*** *bool|null  
 \*/***public function** load($alias)  
{  
 *// protected static $facadeNamespace = 'Facades\\';* **if** (**static**::*$facadeNamespace* && *strpos*($alias, **static**::*$facadeNamespace*) === 0) {  
 $this->loadFacade($alias);  
  
 **return true**;  
 }  
  
 **if** (**isset**($this->**aliases**[$alias])) {  
 **return** *class\_alias*($this->**aliases**[$alias], $alias);  
 }  
}

### 2.3$this->app->booted(function () {

//保存在Application->bootedCallbacks[]数组里

$this->**app**->booted(**function** () {  
 $this->**app**[**'router'**]->getRoutes()->refreshNameLookups();  
 $this->**app**[**'router'**]->getRoutes()->refreshActionLookups();  
});

Application

//注册一个新的“启动”侦听器

**public function** booted($callback)  
{

// **bootedCallbacks 回调数组**  
 $this->**bootedCallbacks**[] = $callback;  
  
 **if** ($this->isBooted()) {  
 $this->fireAppCallbacks([$callback]);  
 }  
}

*/\*\*  
 \* Determine if the application has booted.  
 \*确定应用程序是否已启动  
 \** ***@return*** *bool  
 \*/***public function** isBooted()  
{  
 **return** $this->**booted**;  
}

**//执行回调**

**protected function** fireAppCallbacks(**array** $callbacks)  
{  
 **foreach** ($callbacks **as** $callback) {  
 *call\_user\_func*($callback, $this);  
 }  
}

# 中间件

Laravel用装饰者模式来实现中间件的功能、

通过上面laravel启动流程可以知道中间件的实现是在下面过程中



Laravel一开始在App\Http\Kernel中定义了$middleware, $middlewareGroups

$routeMiddleware(路由中间件)三个中间件数组.

## 中间件的注册与运行

### 1.定义中间件数组

app\http\kernel类继承了Illuminate\Foundation\Http\Kernel

并定义了$middleware, $middlewareGroups，$routeMiddleware(路由中间件)三个中间件数组.

### 2．实例化app\http\kernel

在获取app\http\kernel实例时$kernel = $app->make(Illuminate\Contracts\Http\Kernel::***class***);

获取实例化app\http\kernel这个类

### 3. 创建一个新的HTTP内核实例

**public function** \_\_construct(Application $app, Router $router)  
{  
 $this->**app** = $app;  
 $this->**router** = $router;  
  
 $router->**middlewarePriority** = $this->**middlewarePriority**;  
  
 **foreach** ($this->**middlewareGroups as** $key => $middleware) {  
 $router->middlewareGroup($key, $middleware);  
 }  
  
 **foreach** ($this->**routeMiddleware as** $key => $middleware) {  
 *//aliasMiddleware存放注册中间件的简写名称的数组* $router->aliasMiddleware($key, $middleware);  
 }  
}

### 4. 运行中间件， 处理传入请求

$response = $kernel->handle(  
 $request = Illuminate\Http\Request::*capture*()  
);

Handle()的处理流程在上面的route启动流程中有提到

*/\*\*  
 \* Send the given request through the middleware / router.  
 \* 通过中间件或路由发送给定的请求  
 \** ***@param*** *\Illuminate\Http\Request $request  
 \** ***@return*** *\Illuminate\Http\Response  
 \*/***protected function** sendRequestThroughRouter($request)  
{  
 $this->**app**->instance(**'request'**, $request);  
 Facade::*clearResolvedInstance*(**'request'**);  
 *//Bootstrap the application for HTTP requests  
 //启动应用程序的http请求  
 //各种服务的注册与启动* $this->bootstrap();  
 **return** (**new** Pipeline($this->**app**))*//请求的分发* ->send($request)  
 ->through($this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**)  
 ->then($this->dispatchToRouter());  
}

请求会通过Pipeline分发请求，pipeline在下面会提到

### pipeline的流程

**return** (**new** Pipeline($this->**app**))*//请求的分发* ->send($request)  
 ->through($this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**)  
 ->then($this->dispatchToRouter());

先介绍pipeline的send、through、then三个方法中参数的意义，send($request)，$request是发送的请求

$this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**

shouldSkipMiddleware（）*确定是否为应用程序禁用了中间件*

**middleware就是app\http|kernel定义的数组**

**protected $middleware** = [  
 \App\Http\Middleware\TrustProxies::***class***,  
 \App\Http\Middleware\CheckForMaintenanceMode::***class***,  
 \Illuminate\Foundation\Http\Middleware\ValidatePostSize::***class***,  
 \App\Http\Middleware\TrimStrings::***class***,  
 \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::***class***,  
 HeaderMiddleware::***class***];

$this->dispatchToRouter()*获取路由分发回调函数*

*具体的流程以后再reposnse中学习？*

**protected function** dispatchToRouter()  
{  
 **return function** ($request) {  
 $this->**app**->instance(**'request'**, $request);  
 **return** $this->**router**->dispatch($request);  
 };  
}

#### 1. send设置通过管道发送的对象

*/\*\*  
 \*设置通过管道发送的对象  
 \** ***@param*** *mixed $passable  
 \** ***@return*** *$this  
 \*/***public function** send($passable)  
{  
 *//passable 正在通过管道的对象* $this->**passable** = $passable;  
 **return** $this;  
}

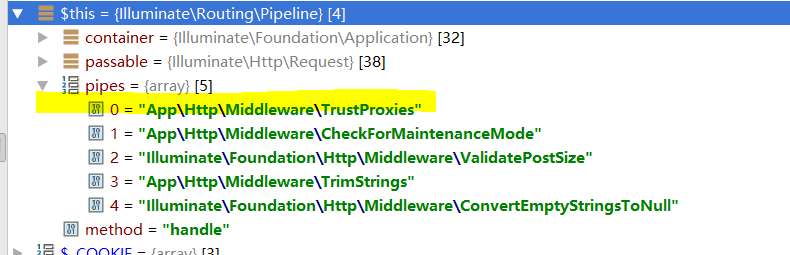
#### 2. through*设置管道数组*

*/\*\*  
 \* 设置管道数组  
 \** ***@param*** *array|mixed $pipes  
 \** ***@return*** *$this  
 \*/***public function** through($pipes)  
{

//*管道类的数组 func\_get\_args*返回一个含有函数参数列表的数组

$this->**pipes** = *is\_array*($pipes) ? $pipes : *func\_get\_args*();  
 **return** $this;  
}

//$pipe初始化



#### 3.then使用最终目标回调运行管道

**public function** then(Closure $destination)  
{  
 $pipeline = *array\_reduce*(  
 *array\_reverse*($this->**pipes**), $this->carry(), $this->prepareDestination($destination)  
 );  
  
 **return** $pipeline($this->**passable**);  
}

*array\_reverse反转数组元素，*

*array\_reduce将*$this->prepareDestination($destination)作为第一个参数，$this->**pipes数组中的第一个元素作为第二个参数调用**$this->carry()并将返回结果再次作为第一个参数，$this->**pipes数组下一个元素作为第二个元素执行**$this->carry()如此反复

##### $this->carry()流程

调用 Illuminate\Routing\Pipeline类,他继承了Illuminate\Pipeline\PipeLine

**protected function** carry()  
{  
 **return function** ($stack, $pipe) {  
 **return function** ($passable) **use** ($stack, $pipe) {  
 **try** {  
 $slice = **parent**::*carry*();  
  
 $callable = $slice($stack, $pipe);  
  
 **return** $callable($passable);  
 } **catch** (Exception $e) {  
 **return** $this->handleException($passable, $e);  
 } **catch** (Throwable $e) {  
 **return** $this->handleException($passable, **new** FatalThrowableError($e));  
 }  
 };  
 };  
}

Illuminate\Pipeline\PipeLine

**protected function** carry()  
 {  
 **return function** ($stack, $pipe) {  
 **return function** ($passable) **use** ($stack, $pipe) {  
 **if** (*is\_callable*($pipe)) { *//判断$pipe是否可调用* **return** $pipe($passable, $stack);  
 }  
 **elseif** (! *is\_object*($pipe)) {  
 [$name, $parameters] =$this->parsePipeString($pipe);$pipe = $this->getContainer()->make($name);  
 $parameters = *array\_merge*([$passable, $stack], $parameters);  
 }  
 **else** {$parameters = [$passable, $stack];  
 }  
 *//相当于闭包装饰者的handle()* $response = *method\_exists*($pipe, $this->**method**)  
 ? $pipe->{$this->**method**}(...$parameters)  
 : $pipe(...$parameters);  
  
 **return** $response **instanceof** Responsable  
 ? $response->toResponse($this->getContainer()->make(Request::***class***))  
 : $response;  
 };  
 };  
 }

**method=”handle”**

$response = *method\_exists*($pipe, $this->**method**)  
 ? $pipe->{$this->**method**}(...$parameters)  
 : $pipe(...$parameters);  
具体的流程和下面的闭包装饰者一样，相当于闭包装饰者自动化代码中的handle（）

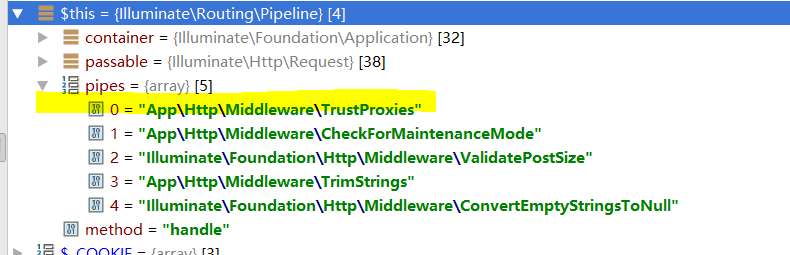
即去调用中间件类中的handle()

比如：

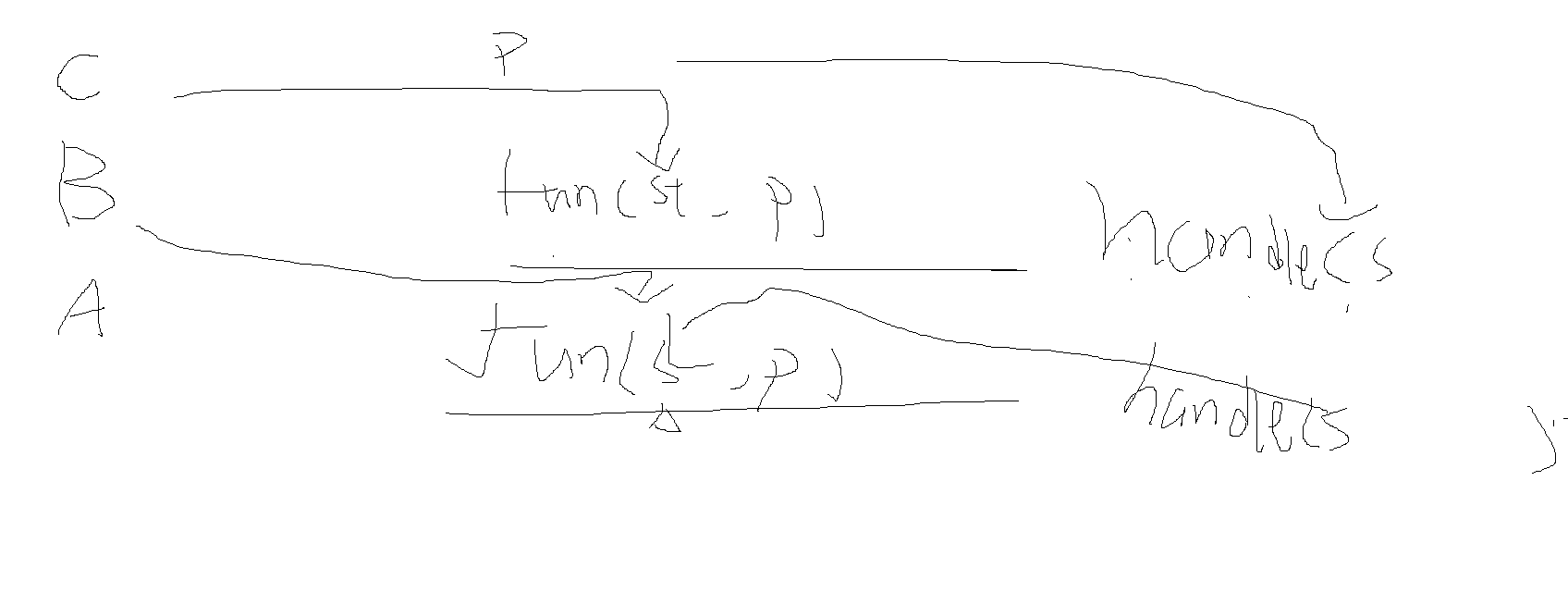
$pipe=App\Http\Middleware\TrustProxies

则调用他的handle()

**public function** handle(Request $request, Closure $next)  
{  
 $request::*setTrustedProxies*([], $this->getTrustedHeaderNames()); *// Reset trusted proxies between requests* $this->setTrustedProxyIpAddresses($request);  
 **return** $next($request);  
}



具体的流程：



## 闭包装饰者

在上面laravel启动流程中

$response = $kernel->handle(  
 $request = Illuminate\Http\Request::*capture*()  
);

会触发

**protected function** sendRequestThroughRouter($request)  
{  
 $this->**app**->instance(**'request'**, $request);  
 Facade::*clearResolvedInstance*(**'request'**);  
 *//Bootstrap the application for HTTP requests  
 //启动应用程序的http请求  
 //各种服务的注册与启动* $this->bootstrap();  
 **return** (**new** Pipeline($this->**app**))*//请求的分发* ->send($request)  
 ->through($this->**app**->shouldSkipMiddleware() ? [] : $this->**middleware**)  
 ->then($this->dispatchToRouter());  
}

在这段代码中中间件处理由Pipeline完成，它是一个闭包装饰者。Request是具体类，相当于上面的caffee,middleware中间件是装饰者类相当于上面的dressing

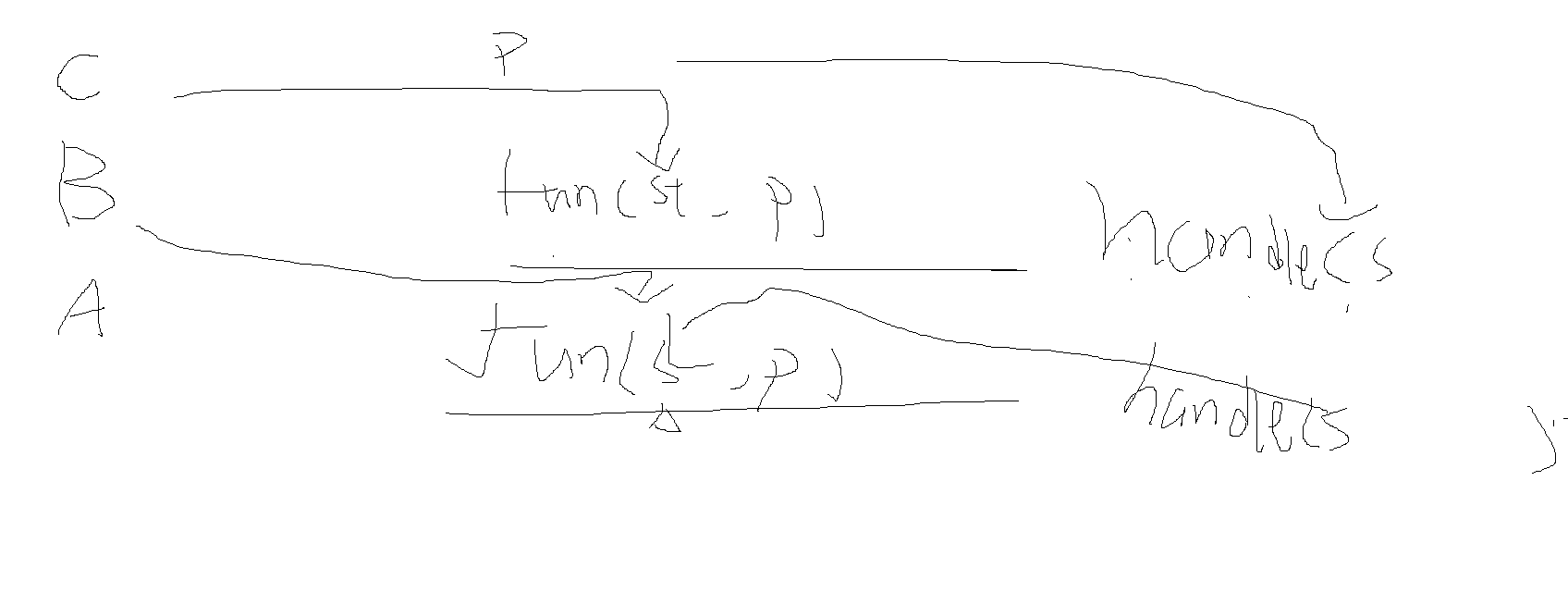
## Pipeline

参考

<https://www.php.cn/phpkj/laravel/414215.html>

laravel源码 pipeline Pipeline中间件处理源码分析

在理解pipeline之前了解闭包装饰者 (见设计模式/结构模式/闭包装饰者代码)



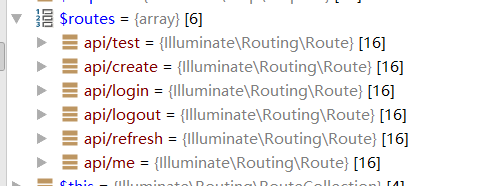
Laravel的中间件由Pipeline来完成，采用闭包装饰者模式实现中间件。

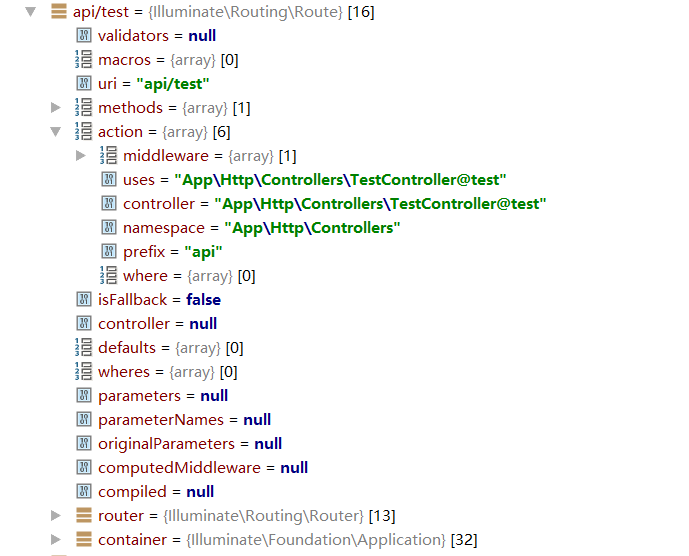
Request 是具体类 相当于代码中的 Starbucks类

Middleware中间件是装饰者类相当于Decorator，一个个具体的中间件就相当于 Whip类和Mocha类。

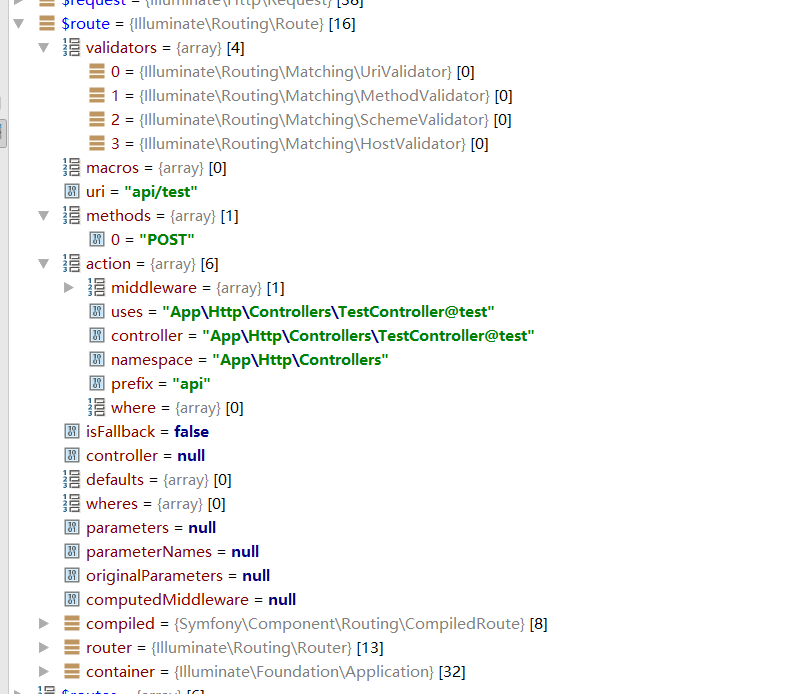
# 路由中间件

$routes





$route



# ArrayAccess(数组式访问)

Laravel源码中有很多arrayaAccess，那么这到底是个啥呢。

参考：<https://www.cnblogs.com/foreverno9/p/8640232.html>

ArrayAccess接口可以让对象可以像数组一样访问如this ['name'] ，在PHP5中加入.

这个接口定义了4个必须实现的方法

public function offsetExists($offset);

public function offsetGet($offset);

public function offsetSet($offset, $value);

public function offsetUnset($offset);

$foo['xxx'] 对应调用offsetGet方法。

$foo['xxx'] = 'yyy' 对应调用offsetSet方法。

isset($foo['xxx']) 对应调用offsetExists方法。

unset($foo['xxx']) 对应调用offsetUnset方法。

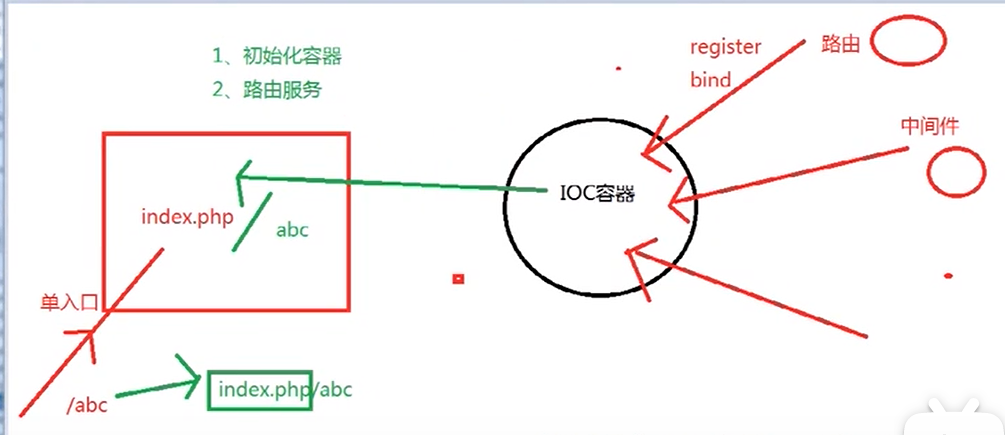
# 响应

# 系统运行方式

系统组件（路由、中间件、service）

应用组件

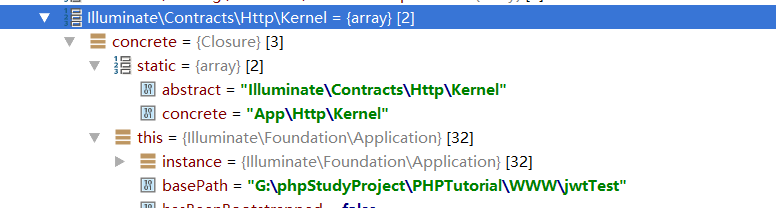
组件注册在ioc容器中



运行步骤：

入口文件Index.php进入

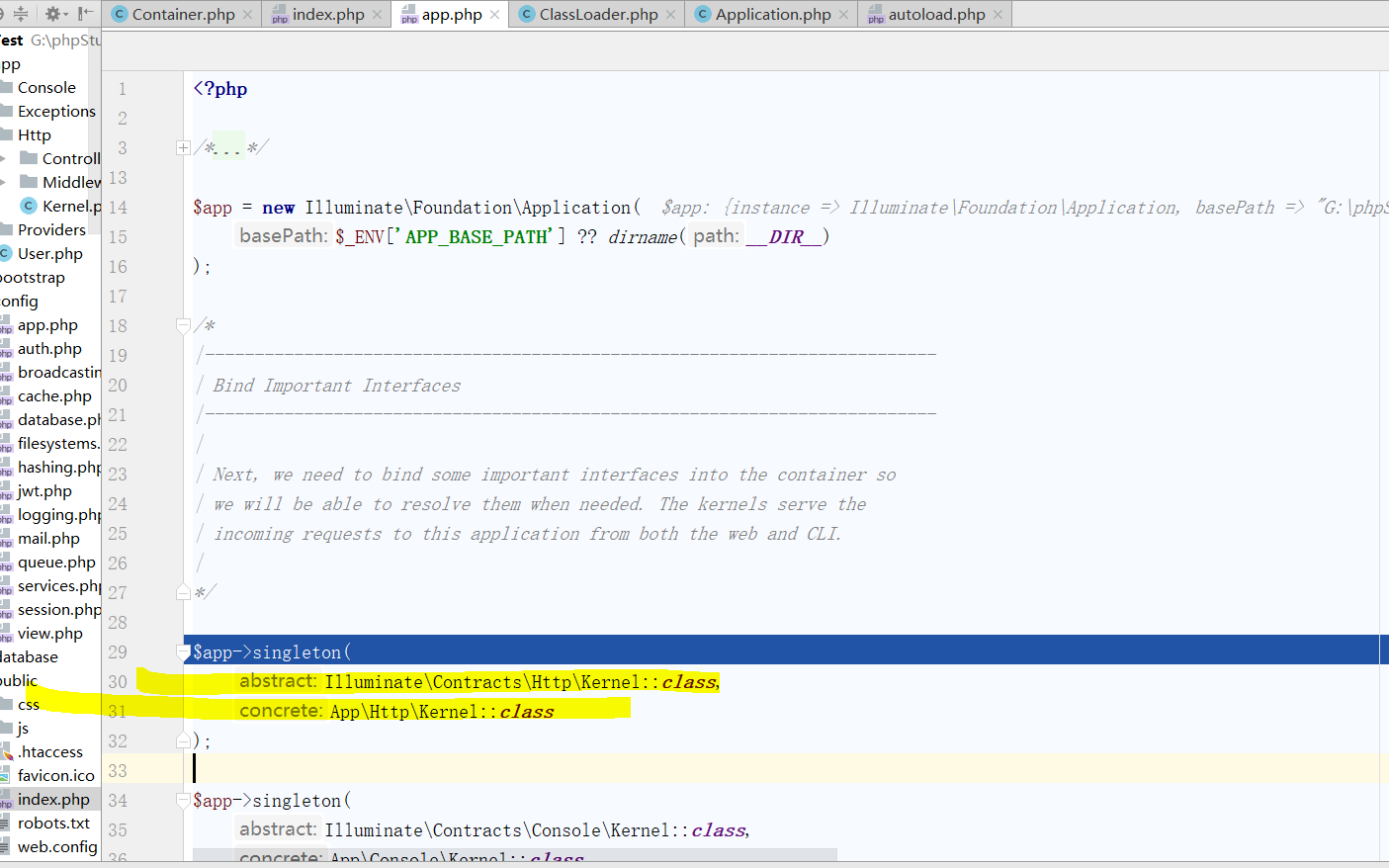
1. 初始化容器
2. 路由服务



什么时候绑定的

# AOP

　AOP为 Aspect Oriented Programming 的缩写,即面向切面编程, 通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术.。AOP是OOP的延续, 是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离, 从而使得业务逻辑各部分之间的耦合度降低, 提高程序的可重用性, 同时提高了开发的效率。



在这时候绑定