跟踪symfony3记录

1. 自动加载

自动加载完成后，ClassLoader：：loadClass 还有 swift里也有一个自动加载得函数。

1. Debug

调用Debug::enable();

ExceptionHandler::register();

set\_exception\_handler(array($handler, 'handle'))

ErrorHandler::register(new ErrorHandler(new BufferingLogger()));

$this->loggedErrors;是记录得错误，$this->loggers;是记录得方式

register\_shutdown\_function(\_\_CLASS\_\_.'::handleFatalError');

set\_error\_handler(array($handler, 'handleError'), $handler->thrownErrors | $handler->loggedErrors);

set\_exception\_handler(array($handler, 'handleException'))

public function setExceptionHandler(callable $handler = null)

{

$prev = $this->exceptionHandler;

$this->exceptionHandler = $handler;

return $prev;

}

DebugClassLoader::enable();

总结：注册错误处理函数，开启调试，把自动加载接入调试里。

3、

class AppKernel extends Kernel

abstract class Kernel implements KernelInterface, RebootableInterface, TerminableInterface

interface KernelInterface extends HttpKernelInterface, \Serializable

interface HttpKernelInterface

$this->checkClass($class, $file);

$deprecations = $this->checkAnnotations($refl, $name);

$parentAndOwnInterfaces = $this->getOwnInterfaces($class, $parent);

1. $kernel = new AppKernel('dev', true);

public static function createFromGlobals()

$request = self::createRequestFromFactory($\_GET, $\_POST, array(), $\_COOKIE, $\_FILES, $server);

$response = $kernel->handle($request);

$this->initializeBundles();

public function registerBundles()

实例化bundle经过的类：

new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),

class FrameworkBundle extends Bundle

abstract class Bundle implements BundleInterface

use Symfony\Component\DependencyInjection\ContainerAwareTrait;

trait ContainerAwareTrait

interface BundleInterface extends ContainerAwareInterface

interface ContainerAwareInterface

new Symfony\Bundle\SecurityBundle\SecurityBundle(),

class SecurityBundle extends Bundle

new Symfony\Bundle\TwigBundle\TwigBundle(),

$this->initializeContainer();

$cache = new ConfigCache($cacheDir.'/'.$class.'.php', $this->debug);

{

$checkers = array(new SelfCheckingResourceChecker());

parent::\_\_construct($file, $checkers);

}

class ConfigCache extends ResourceCheckerConfigCache

{

$this->resourceCheckers = $resourceCheckers;

}

class ResourceCheckerConfigCache implements ConfigCacheInterface

interface ConfigCacheInterface

加载meta文件？？？

class ComposerResource implements SelfCheckingResourceInterface, \Serializable

class SelfCheckingResourceChecker implements ResourceCheckerInterface

interface ResourceCheckerInterface

class appDevDebugProjectContainer extends Container

class Container implements ResettableContainerInterface

interface ResettableContainerInterface extends ContainerInterface

interface ContainerInterface extends PsrContainerInterface

$container = $this->buildContainer();

$container = $this->getContainerBuilder();

class ContainerBuilder extends Container implements TaggedContainerInterface

interface TaggedContainerInterface extends ContainerInterface

class ParameterBag implements ParameterBagInterface

（将bundle放到 $this->parameters[$this->normalizeName($name)] = $value;）

$container->addObjectResource($this);

$this->prepareContainer($container);（实例化bundle里面的extension）

$this->resources[(string) $resource] = $resource（E:\wnmp\nginx\html\symfony3\app\AppKernel.php）;

$this->extensions[$extension->getAlias()] = $extension;

到配置文件了：

解析import：

$this->parseImports($content, $path);

$this->registerClasses($definition, $namespace, $service['resource'], $exclude);

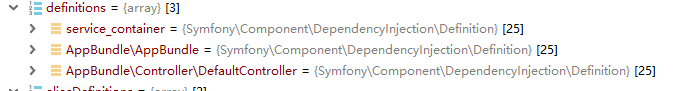
service配置文件的时候，加载了src/controller下面的文件

并加到container的resource里面

现在看看container里面有什么？



分析出servic里面的，放在definition里面



class AddAnnotatedClassesToCachePass implements CompilerPassInterface

public function \_\_construct(Kernel $kernel)

{

$this->kernel = $kernel;

}

接下来到这里了

$container->compile();

过程：

$compiler = $this->getCompiler();

$compiler->compile($this);

###############

foreach ($this->passConfig->getPasses() as $pass) {

$pass->process($container);

}

疑问一：

Container里面属性的作用。和如何初始化（怎么来的）。

疑问二：

Container的extensions是怎么来的？

这段代码太繁杂了，先放着先。看得老夫一口老血。。。。

Kernel文件的如下函数

$this->dumpContainer($cache, $container, $class, $this->getContainerBaseClass());

生成了一系列的自动生成文件，并且替换了kernel里面的container属性。（这个非常重要，要留意）

重头戏：

**return** $this->getHttpKernel()->handle($request, $type, $catch);

Httpkernel.php:

return $this->handleRaw($request, $type);

class GetResponseEvent extends KernelEvent

class KernelEvent extends Event

注意到一件事情：listener，事件监听这个东西。

$this->preProcess($eventName);

以下：开启section？？？

$this->preDispatch($eventName, $event);

执行每个事件监听。

public function dispatch($eventName, Event $event = null)

Symfony\Component\HttpKernel\EventListener\FragmentListener

这里是url和baserul 设置到request里面

每个listenner共用一个stopwatch

$this->**context**->fromRequest($request);

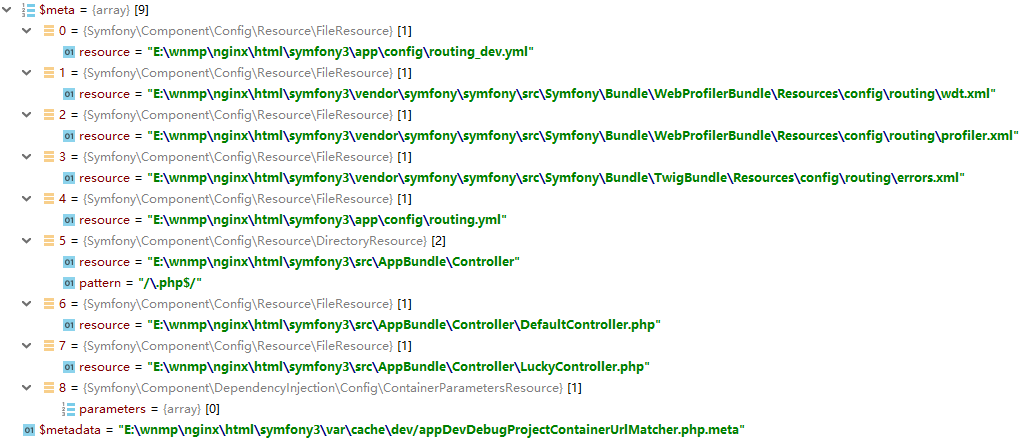
留意一下RouterListener.php下面的context，这里面会有各种route信息（RequestContext实例）

Matcher是router实例也留意一下。

跟新写入是如下代码：

**function** (ConfigCacheInterface $cache) {  
 $dumper = $this->getMatcherDumperInstance();  
 **if** (*method\_exists*($dumper, **'addExpressionLanguageProvider'**)) {  
 **foreach** ($this->**expressionLanguageProviders as** $provider) {  
 $dumper->addExpressionLanguageProvider($provider);  
 }  
 }  
  
 $options = **array**(  
 **'class'** => $this->**options**[**'matcher\_cache\_class'**],  
 **'base\_class'** => $this->**options**[**'matcher\_base\_class'**],  
 );  
  
 $cache->write($dumper->dump($options), $this->getRouteCollection()->getResources());  
}

检测跟新是检测如下文件：



引入了E:\wnmp\nginx\html\symfony3\var\cache\dev\appDevDebugProjectContainerUrlMatcher.php

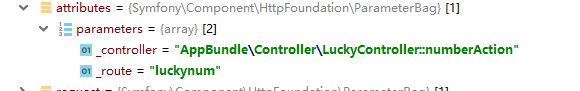
文件，（问题：什么时候产生的？这个问题非常重要）我改了一下url可以发现文件的修改时间改变了，可以追踪一下。

$matcher = $this->getMatcher();

这个就是返回上述文件产生的类。

Routerlisterner.php下面的match是router怎么来的。

Request里的**attributes多了两个属性**



Listerner里面最后一个看不懂。。。先放着

获取controller可以发现，

if (isset($this->fileMap[$id]) || isset($this->methodMap[$id])) {

return true;

}

里面：

$this->fileMap[$id]有他的文件，getLuckyControllerService.php

改变controller里面的$controller->setContainer($this->**container**);

Container。

到这里有几个问题需要思考：

1. 为什么说bundle是一等公民，是如何体现的？
2. 编译container，匹配路由，编译urlmatch需要整理一下（留意一下container里面的services是怎么来的，事件的dispatch是怎么来的）
3. 验证一下是否container可以重用。
4. 需要重新熟悉一下这个框架
5. 再次整理一下bundle，container，service等等
6. 数据库那方面。
7. 从设计方面再次理解这个框架。
8. 依赖注入的实现，lazyload这个看一下。
9. 还要看一下symfony的文件，了解它的功能架构。

Service是在一步一步增多。

判断75个文件是否需要更新。

编译后的container内容：

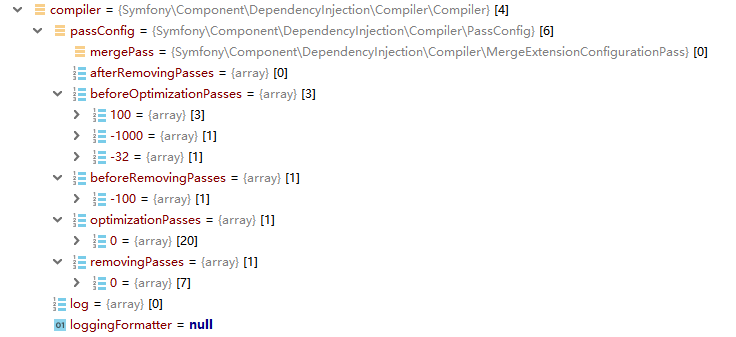


**return** $this->getHttpKernel()->handle($request, $type, $catch);

这个语句之后，有一些相关的事件监听，container里面service的增加。

GetResponseEvent事件里面：

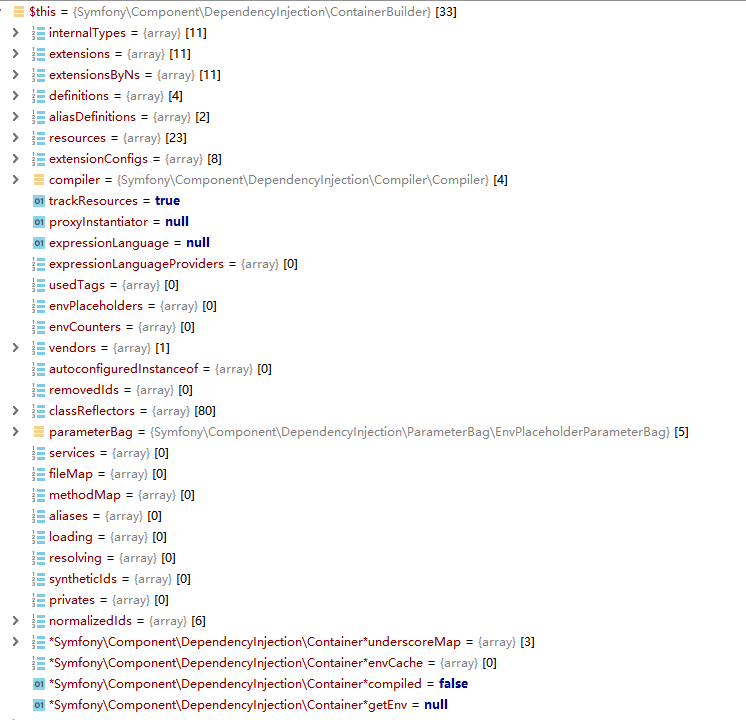
Symfony\Component\HttpKernel\EventListener\RouterListener这个是生成route的url的，有必要看一下。



最初container的样子：



进入compiler钱container的样子：



看注释，compiler是做这些事情：

\* Compiles the container.

\*

\* This method passes the container to compiler

\* passes whose job is to manipulate and optimize

\* the container.

\*

\* The main compiler passes roughly do four things:

\*

\* \* The extension configurations are merged;

\* \* Parameter values are resolved;

\* \* The parameter bag is frozen;

\* \* Extension loading is disabled.

\*

总结：

1. 构建container的时候，是根据definition属性进行编译
2. 标记是否需要重新编译是resource属性的75个文件的内容。

现在看到了httpkernel request这里，下次看url的形成等等路由。现在先小结一个尾巴。

在ContainerKdrximf/appDevDebugProjectContainer.php，里面的一些属性是通过definition的特点，如hastag，private，hotpath，isSynthetic，isShared等等判断。

比如：

methodMap的条件是：

if (!$definition->isSynthetic() && (!$this->asFiles || !$definition->isShared() || $this->isHotPath($definition)))

Filemap的条件是：

if (!$definition->isSynthetic() && $definition->isShared() && !$this->isHotPath($definition))

仔细观察发现：

methodMap是当前文件ContainerKdrximf/appDevDebugProjectContainer.php里的方法

而filemap是下面的getXXXXservice.php文件

在methodmap里面，每个definition下的有一些属性，根据这些属性解析文件。

比如说calls等，判断是否是简单返回。然后再组装方法。

搞清楚这四个变量是干嘛的：

$this->definitionVariables = $this->inlinedDefinitions = null;

$this->referenceVariables = $this->serviceCalls = null;

$this->serviceCalls在reference的时候才会改变

$this->inlinedDefinitions 在argument是definition的时候才会改变。

有file的inliDefinitions才会include文件进来

if ($arguments = array\_filter(array($inlineDef->getProperties(), $inlineDef->getMethodCalls(), $inlineDef->getConfigurator())))

这一句判断是否是简单实例化。

整理一下顺序：

$this->addService($id, $definition)

$code .= $this->addServiceInclude($id, $definition);

$code .= $this->addInlineService($id, $definition);

$code .= $this->addServiceInstance($id, $definition, $isSimpleInstance);

$this->addInlineVariables($id, $definition, $arguments, false)

$code .= $this->addServiceProperties($inlineDef, $name);

$code .= $this->addServiceMethodCalls($inlineDef, $name);

$code .= $this->addServiceConfigurator($inlineDef, $name);

在addInlineService里面：

可以看到definition下面的call，在addServiceInstance这里dump出来，

如果里面是definition，继续调用addInlineService，然后在addServiceMethodCalls里面实现调用。

如果definition里面有argument，里面是实例化的参数

if ('instance' === $name) {

$code .= $this->addServiceInstance($id, $definition, $isSimpleInstance);

} else {

$code .= $this->addNewInstance($inlineDef, '$'.$name, ' = ', $id);

这两个是实例化的dump

根据条件可以知道，生成函数的是满足除了下述条件：

if ($definition->isSynthetic() || ($this->asFiles && $definition->isShared() && !$this->isHotPath($definition))) {

continue;

}

可以看到，这个条件就是methodmap的条件。

生成文件的条件是：

if (!$definition->isSynthetic() && $definition->isShared() && !$this->isHotPath($definition))

$this->inlinedDefinitions是根据这个去生成include文件的。

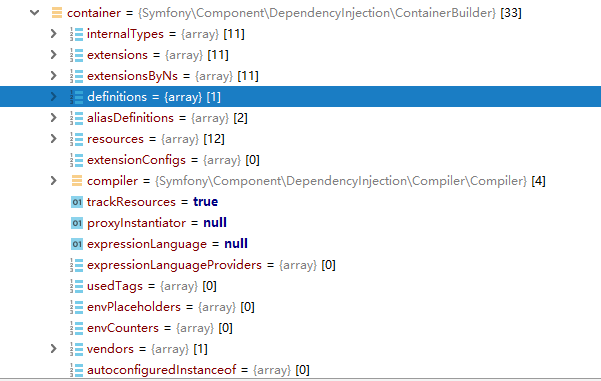
文件的生成跟方法的生成大同小异，接下来看一下

1. 容器container里的definition和resource是怎么来的，搞清楚为什么是这些文件要作为resource和definition。

2、路由文件及其相关

Container的形成：

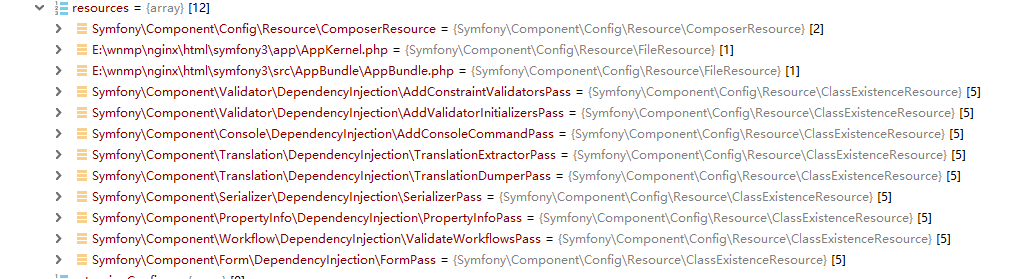
重点观察definition和resource的形成。



上述的definition和resources是在

$container = $this->buildContainer();load配置文件之前的内容。

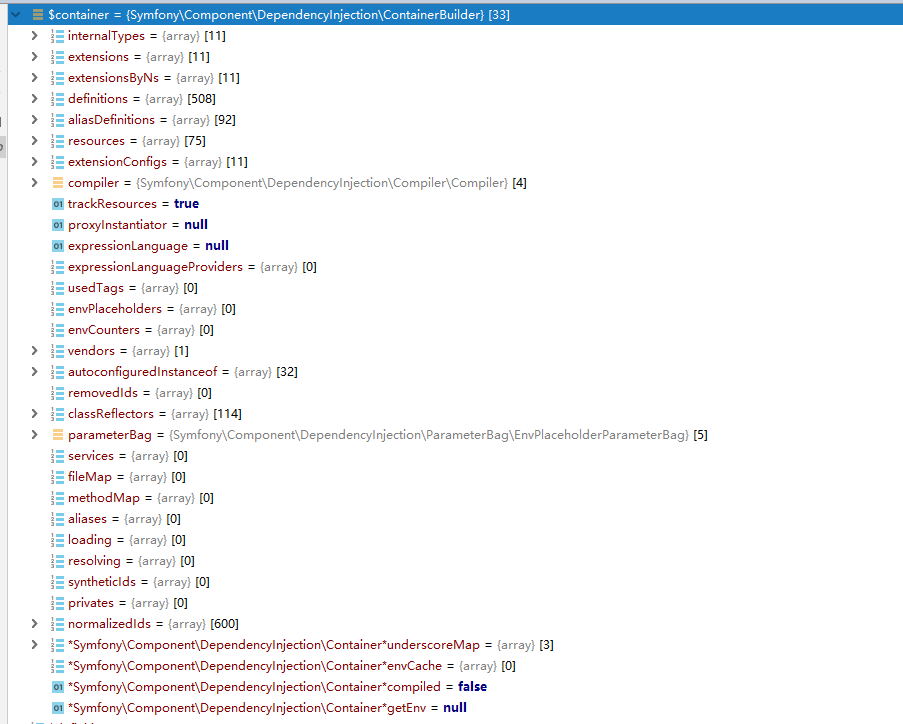
Resources的内容：



Definition的内容：



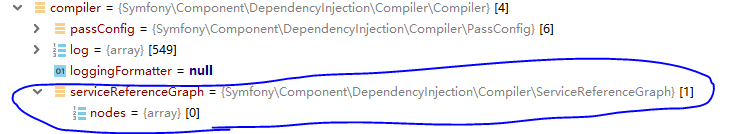
第一个pass，mergeExtentionConfigurationPass后，container的内容是这样的：



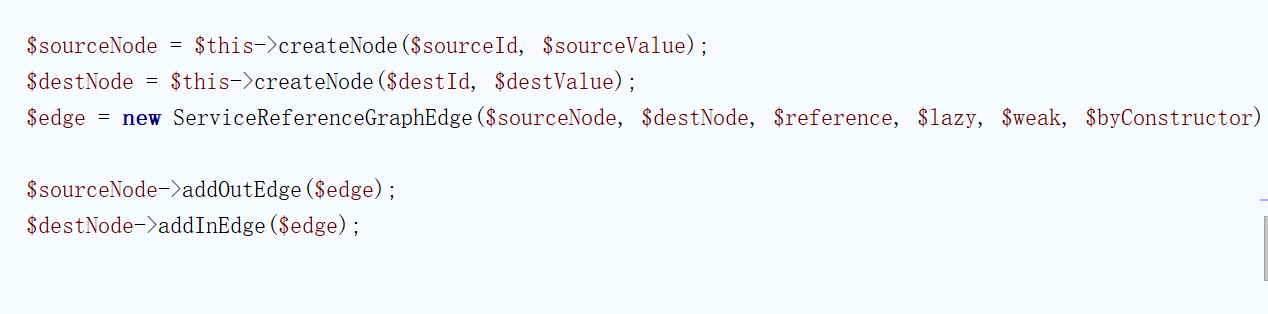
Definition比较多，resources刚刚好。

Process看到26个extensionpass，ide极度卡顿

经过完全部pass之后，definition变为281个，还有，留意一下



这个东西。



由上图可知，sourcenode是增加edge的外节点，destnode是增加edge的内节点

在checkcirclepass里面，所作的是获取每一个node的外节点加入到$this->**checkedNodes**[$id] = **true**;

，

明天从79开始看起

内联服务，类比内联函数

Graph里的node如何产生？

Createnode的时候

什么时候createnode呢？

分析一个definition的时候，它有reference的时候，就会调用create。

也就是说，被调用的reference节点都有一个inline。Ids是这个inline的resourceid。这个是当初的调用者。

Definition减少是在RemoveUnusedDefinitionsPass里面。如果node里面没有这个id，前提是即不是public也不是private的definition。

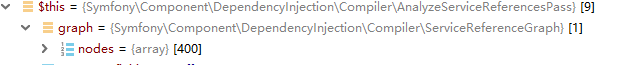
有几个概念不懂：

Inlined service，在InlinedServiceDefinitionsPass这里有判断是否是内联服务并写入log文件，并存储在$this->**inlinedServiceIds**[$id][] = $this->**currentId**;

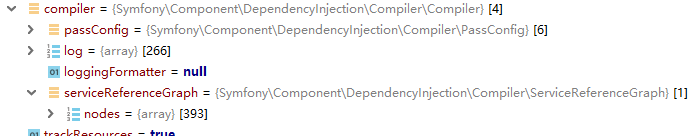
里面。

现在观察一下graph里面node的变化

最开始:

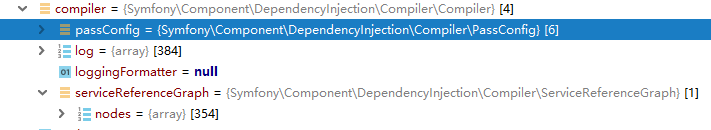


analyze之后:

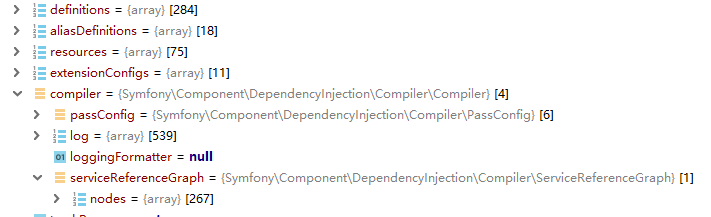


编译的时候,内联服务(inlined service是把原来的referenc变成definition),只要搞清楚这个在编译的时候有什么不一样,就可以知道inlined service这个概念了

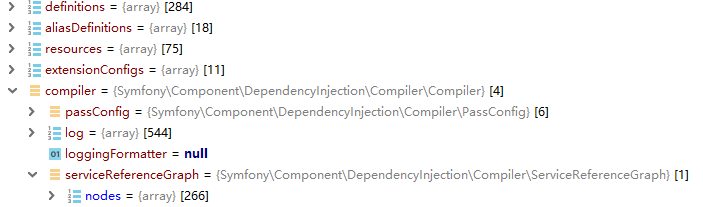
再次分析:



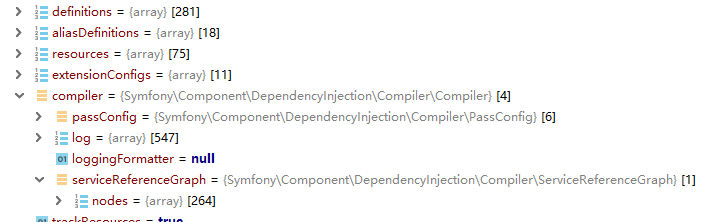
第二次循环第一次anaize:



第二次循环第二次:



最后一次:



Warmup ，各个bundle的boot之后，到接收请求对象

事件监听：

Kernel.request有八个事件监听：

1. Symfony\Component\HttpKernel\EventListener\DebugHandlersListener

大概处理debug的，重置errorhandle的操作。

1. Symfony\Component\HttpKernel\EventListener\ValidateRequestListener

处理request

1. Symfony\Component\HttpKernel\EventListener\SessionListener

设置session

1. Symfony\Component\HttpKernel\EventListener\FragmentListener

Do nothing.

1. Symfony\Component\HttpKernel\EventListener\RouterListener
2. Symfony\Bundle\FrameworkBundle\EventListener\ResolveControllerNameSubscriber
3. Symfony\Component\HttpKernel\EventListener\LocaleListener
4. Symfony\Bundle\SecurityBundle\Debug\TraceableFirewallListener