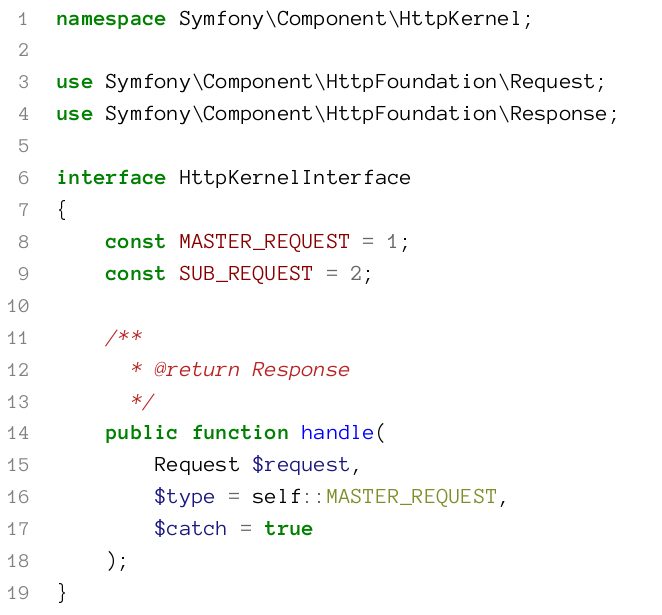
**I 从请求到响应的旅程**

# 1. HttpKernelInterface

1. Symfony的HttpKernelInterface非常有名：



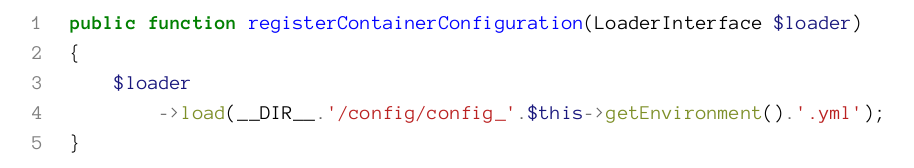
　　此接口的实现只要实现一个方法，从而声明自己具备将请求以某种形式转换为响应的能力。当你看一看Symfony项目的/web目录里的前面的控制器，你会看到这个handle()方法在处理web请求里扮演了一个核心角色，可能如你预期的那样：



　　首先，AppKernel实例化。这是一个特定给你项目的类，你可以在/app/AppKernel.php找到它。它允许你登记你的包，以及修改一些主要设定，如缓存目录的位置或者应该加载的配置文件。它的构造器参数是环境的名称以及内核是否应该在debug模式运行。

环境

环境可以是任意字符串。它的主要用途是判断应该加载哪个配置文件（如　config\_dev.yml或config\_prod.yml）。这在AppKernel里已经得到了明确：



**Debug模式**

在debug模式你将拥有：

* 一个漂亮的，冗长的异常页，为debug掉问题显示全部需要的信息。
* 冗长的错误消息不会在漂亮的异常页面里渲染出来。
* 程序不同的部分运行时所需的时间的详尽信息（启动，数据库调用，模板渲染，等等）。
* 关于请求的大量信息（使用web profiler和如影随形的toolbar）
* 自动缓存失效：这将确保修改config.yml，routing.yml以及类似的其他配置在没有重新编译整个服务容器或每个请求的路由匹配器的情况下将被顾及到（可能会花一些时间）。

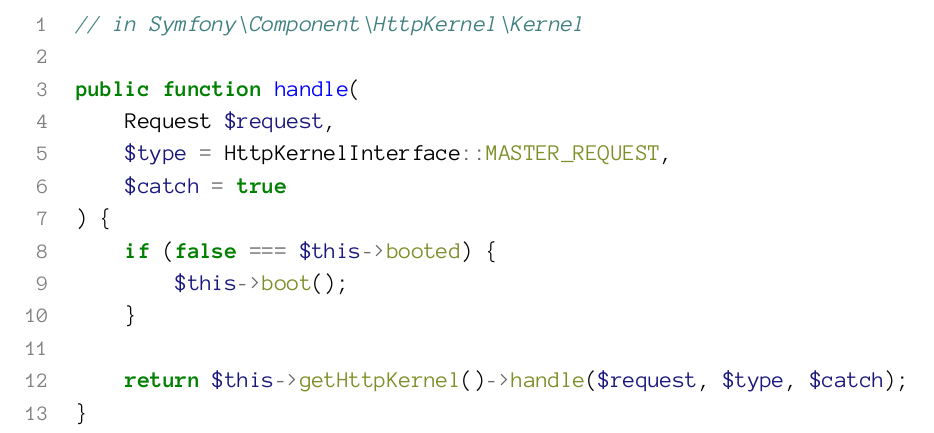
接下来有一个Request对象是基于现有的PHP超级全局变量创建的（$\_GET, $\_POST, $\_COOKIE, $\_FILES和$\_SERVER）。Request类与其他同样来自HttpFoundation组件的类共同提供了面向对象的方法来包装超级全局变量。这些类也涵盖了很多独有方法，在不同的PHP版本或不同平台你或许曾经用到过。在Symfony环境里使用Request来获取你通常使用超级全局变量来获得的数据是很明智的选择。

然后，AppKernel实体的handle()方法被调用了。它唯一的参数就是当前这个Request对象。（“master”）请求默认参数以及是否获取处理异常（yes）将被自动添加。

这个handle()方法的结果得保证是一个Response实例（也来自HttpFoundation组件）。最后，响应（response）将被返回给发起请求的客户端，比如一个浏览器。

# 1.1 启动kernel

当然，这个魔法发生在内核的handle()方法里。你将发现这个方法实现了Kernel类，其是AppKernel的父类：

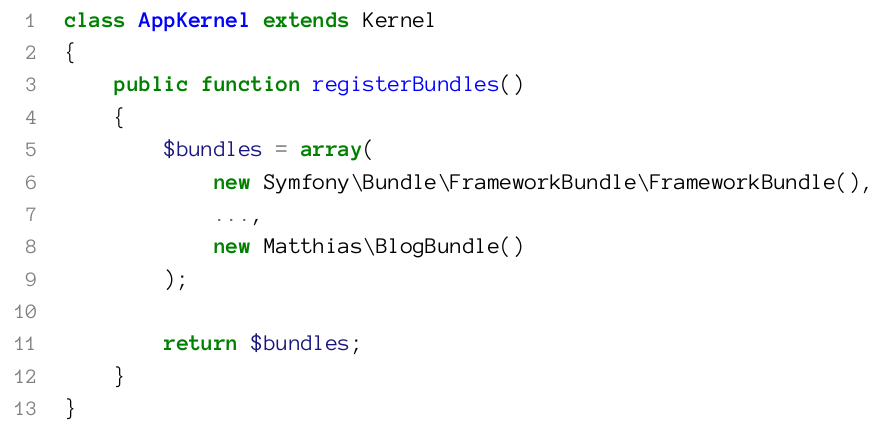


首先，它确保了在HttpKernel被请求做其余事之前Kernel启动了。这个启动过程包括：

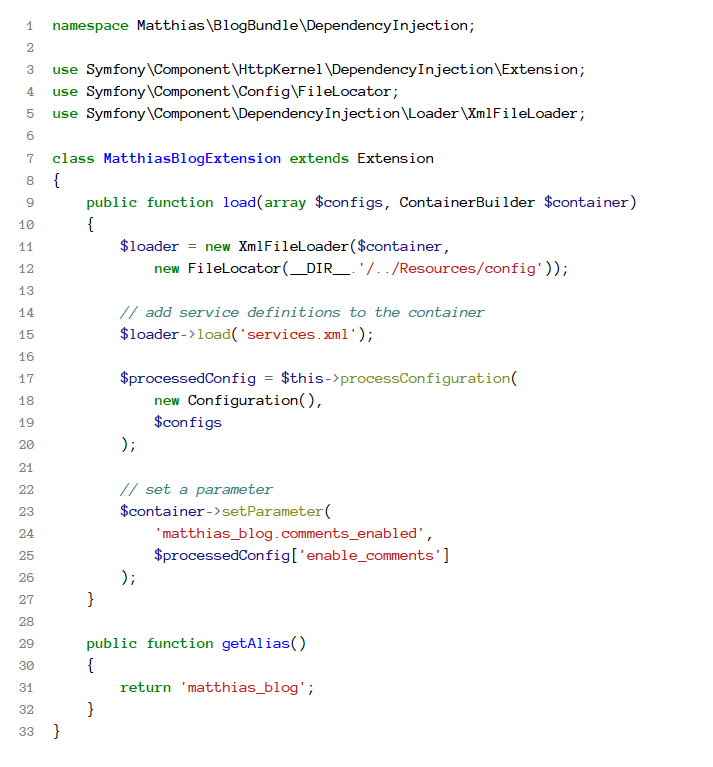
* 初始化全部已登记bundle
* 初始化服务容器

# 作为容器扩展的Bundle

在Symfony开发者看来Bundle就是存放自己代码的地方。每个bundle都应该有一个名字，其反映了在此bundle里应该做些什么类型的事。比如，你可能有一个BlogBundle，一个CommunityBundle，一个CommnentBundle，等等。你在AppKernel.php登记你的bundle，添加它们到现有的bundle列表里：



这绝对是个好主意，它允许通过一行代码就向你的项目插入和提取功能。然而，当查看Kernel是怎样处理全部bundle的，包括你自己的bundle，显而易见bundle主要是使用扩展服务容器的方式，而不是作为代码库的方式。这也就是为什么你会在很多bundle里找到Dependency Injection（依赖注入）目录的原因，并伴有一个{nameOfTheBundle}Extension类。在初始化服务容器处理期间，每个bundle都允许登记一些其自身的服务到服务容器，或许也会添加一些参数，并且可能在容器编译并倾倒到缓存目录之前修改一些服务定义：



一个容器扩展的getAlias()方法返回的名称实际上就是一个键，在它下边你可以设置配置值（比如在config.yml）：



你可以在依赖注射模式了解更多关于bundle配置。

 **每个配置key对应到一个bundle**

在上边的例子里你看到了matthias\_blog是涉及到MatthiasBlogBundle设定的配置key。现在如果告诉你所有你所知的config.yml下key都是这种情况你或许就不会那么吃惊了，比如：framework下的值是涉及到FrameworkBundle的，security下边的值（即使它们定义在一个叫security.yml的分离的文件里）涉及到SecurityBundle。就这么简单！

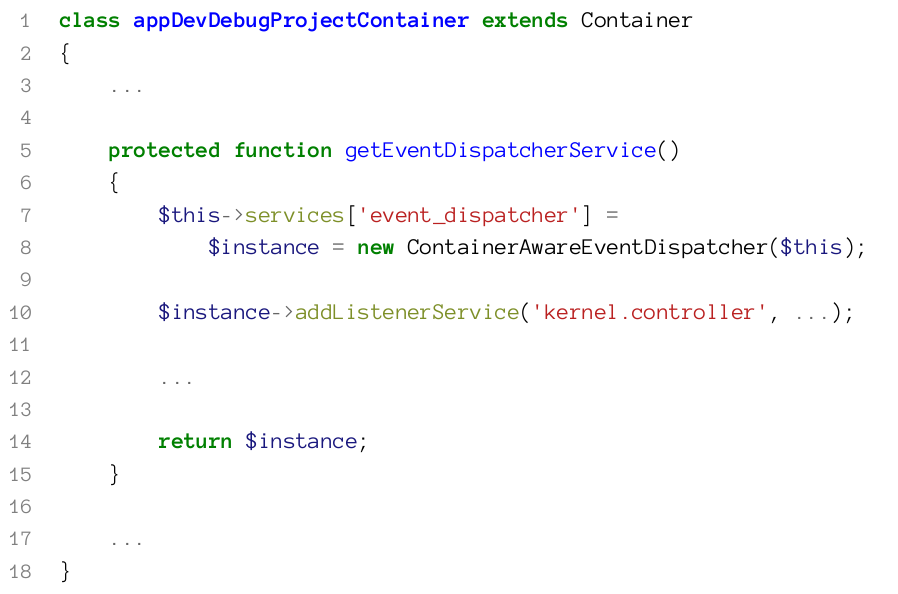
# 创建服务容器

在所有bundle都能够添加它们的服务和参数后，服务容器会完成一个称之为“汇编”的过程。在这个过程中仍可以进行一些最后时刻的对服务定义或参数的修改。此刻也是对服务定义做验证和优化的绝佳时机。之后，容器就处于最终状态了，它会倾倒进两种不同的格式：一份解决了所有定义和参数的XML文件和一份用来在你程序里作为一个也是唯一个服务容器的PHP文件。

两个文件都可以在对应内核环境的缓存目录里找到，比如 /app/cache/dev/appDevDebugProjectContainer.xml。XML文件就如同任何其他正规XML服务定义文件，只是更大一些而已：

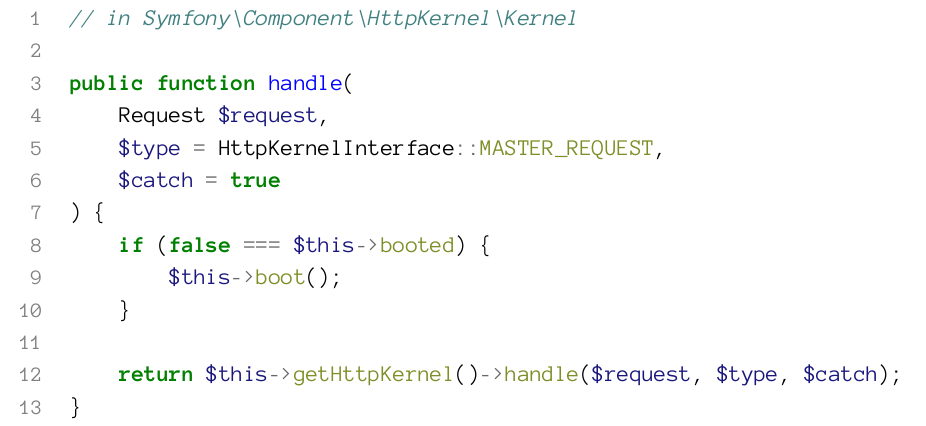


PHP文件为每个可供请求的服务包含了一个方法。任何创建逻辑，像实例化后控制器参数或方法调用都可以在这个文件找到，因此它是debug你服务定义里出现的错误的最理想的地方：

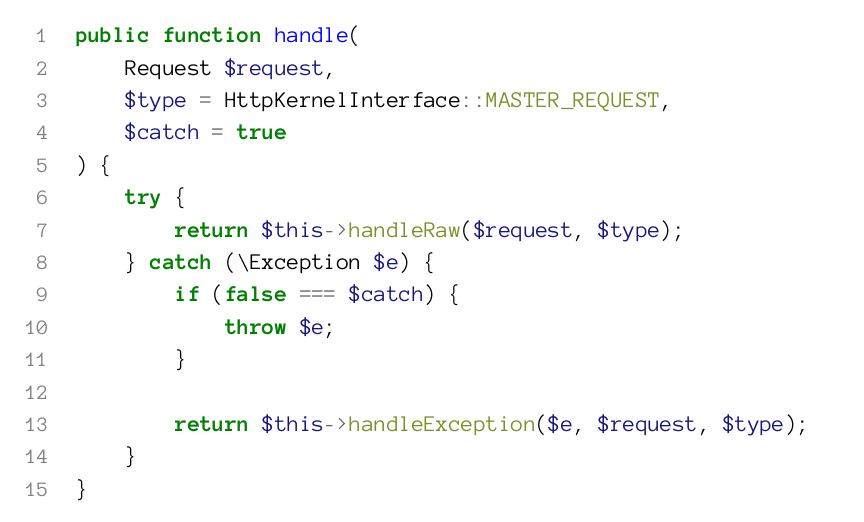


# 1.2 从Kernel到HttpKernel

现在内核已经启动了（如 全部bundle都初始化了，它们的扩展也都登记了，服务容器也落实了），真正的请求处理委托给了一个HttpKernel实例来处理：



HttpKernel完成了HttpKernelInterface，它是以转换一个请求成为一个响应见长的。handle()方法看起来像这样：



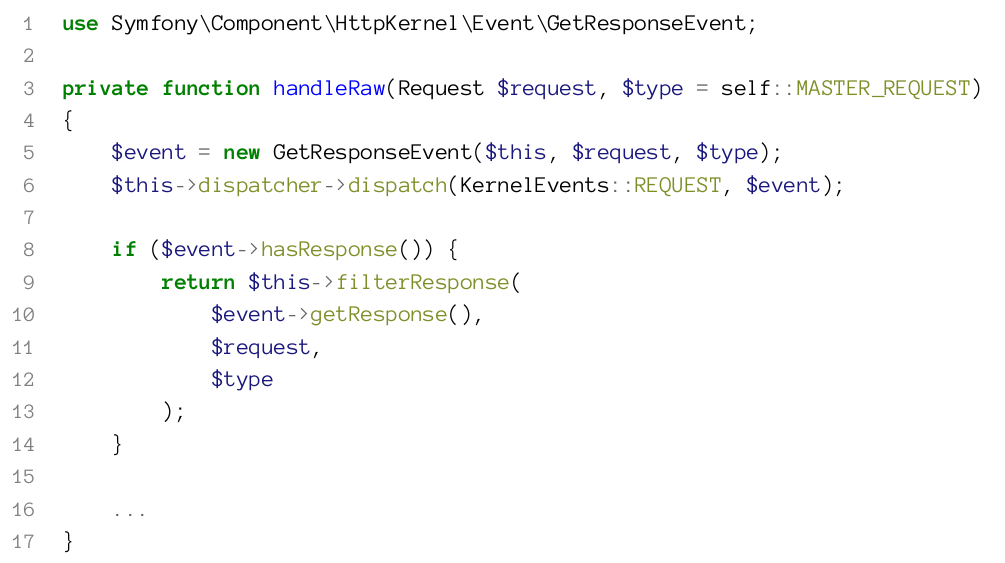
如你所见，大部分的工作都在私有的 handleRaw() 方法里完成，这里 try/catch 语句是获取异常的。当初始化参数 $catch 是 true（其是 “主要” 请求的默认值），每个异常都将被处理的很漂亮。HttpKernel 将仍然尝试寻找谁适合创建 Response 对象（查看 异常处理）。

**2 引导到一个响应的事件**

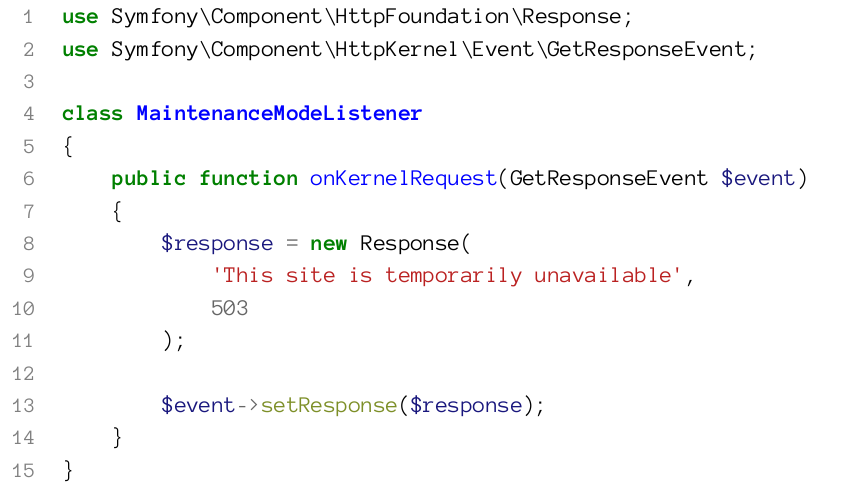
HttpKernel 的 handleRaw() 方法是代码里的一个亮点，在它里边处理一个本身不确定的请求将变得清晰。在里边有所有的方法，你可以用这些方法钩住某过程，并全部替换或仅修改任意中间结果。

**2.1 早期响应**

对此请求加以控制的第一个时刻是一开始的时候。通常 HttpKernel 会尝试通过执行一个控制器来生成一个响应。但任何监听 KernelEvents::REQUEST(kernel.request)事件的事件监听器都被允许来生成一个完全自定义的响应：



如你所见，被创建的事件对象是一个 GetResponseEvent 的实例，它允许监听器用它的 setResponse() 方法来设定一个自定义 Response 对象，例如：

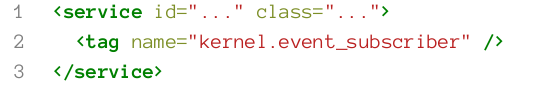


登记事件监听器

HttpKernel 使用的 event dispatcher (事件调度) 也可以通过event\_dispatcher 服务得到。当你想要自动登记一些类作为事件监听器，你可以创建一个服务来定义它，并加入一个 kernel.event\_listener 标签 或 kernel.event\_subscriber ( 这样你要选择实现 EventSubscriberInterface )。



或：



你可以选择给你的事件监听器一个优先级，这样它会比其他监听器优先执行：



数值越高，优先级越高。

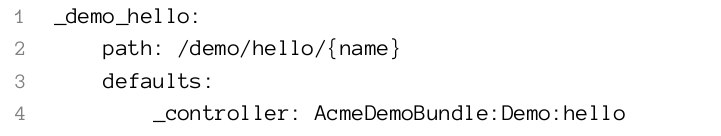
**一些著名的 kernel.request 事件监听器**

框架本身有很多 kernel.request 事件的监听器。这些监听器的大多数都是在内核调用任意控制器之前直接设定一些事情的的。比如一个监听器来确定程序有一个 locale (要么是默认的locale，或者来自 URI 的 \_locale 部分)，另外一个为页面的片段处理请求。

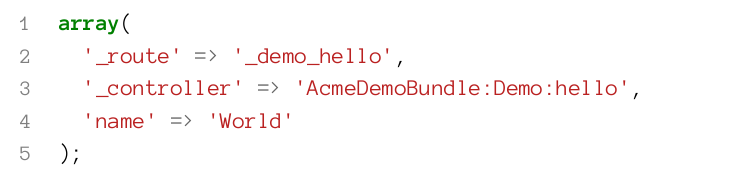
在早期的请求阶段有两个主要“玩家”：RouterListener 和 Firewall。RouterListener 从 Request 对象获取到路径信息，并试着匹配一些已知的路由。它将匹配过程的结果保存在 Request 对象的属性里，例如对应匹配路由的控制器名称：



比如当匹配条件是匹配 /demo/hello/World，路由配置类似这个：



通过调用 match() 返回的参数是定义在 defaults :下的值和路径里占位符动态值 (如 {name}) 的合并:



这些以 Request 参数包结束，被称作“属性”。你或许能猜到：HttpKernel 接下来将检查这些请求属性并执行给定的控制器。

另一个重要的事件监听器是 Firewall（防火墙）。如上所示，RouterListener 并不为 HttpKernel 提供 Response 对象，它只在一个请求的初始阶段做一些事情。相反的，防火墙有时强制生成一个确定的 Response 对象，例如当一个用户没经过必须的验证，因为他请求了一个被保护的页面。防火墙（通过一个复杂的过程）将强制转向到一个比如登陆页面的地方或设置一些需要用户通过使用 HTTP 验证要输入证书的头信息。

**2.2 分解控制器**

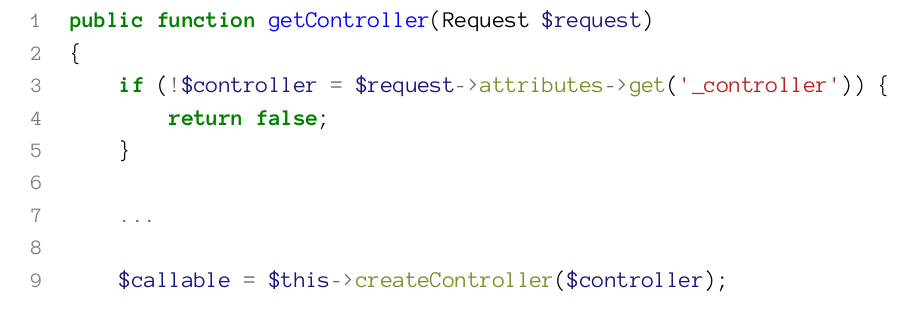
我们看到上边 RouterListener 设定了一个 Request 属性叫做 \_controller，其包含了一些类型的关联到要执行的控制器。这些信息无关 HttpKernel。相反，有一个被称作 ControllerResolver（控制器分解器）的对象被要求来为当前的 Request 对象返回一个控制器：

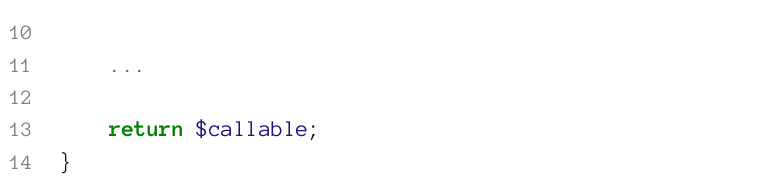


这个解析器本身是一个 ControllerResolverInterface 的实例：



稍后它将被用来判断执行控制器的正确参数。它的首要任务是获得控制器。标准控制器解析器从 Request 属性 \_controller 获取控制器：





因为大多数情况控制器是以一些类型的（速记）标记的字符串，控制器需要实际被创建。

**控制器可以成为的任何东西**

来自 Symfony HttpKernel 组件的 ControllerResolver 支持:

* + - * 可调用数组（对象/方法或类/静态方法的合并）
      * 可调用对象（带有魔法方法 \_\_invoke() 的对象，像匿名函数，其是 \Closure 的一个实例）
      * 可调用对象的类
      * 正规的函数

任何其他作为一个字符串提供的控制器都应该是这个模板 class::method 。

然而，在 FrameworkBundle 里定义的 ControllerResolver 加入了一些额外的控制器名模式：

* + - * BundleName:ControllerName:actionName
      * service\_id:methodName

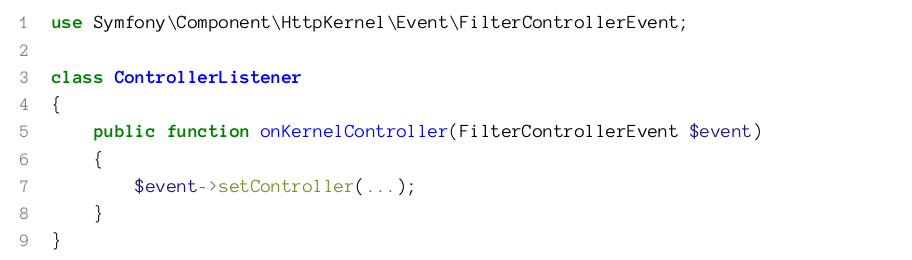
在创建一个控制器的实例后，这个特定的 ControllerResolver 也会检查控制器是否是一个 ContainerAwareInterface 的实例，如果是它会调用 setContainer() 以服务容器提供给此控制器。这也是为什么标准控制器已经提供了服务容器。

**2.3 允许的控制器的替换者**

回到 HttpKernel 控制器现在已经完全可得了，并且也几乎准备好被执行了。虽然即使控制器解析器已经用它的力量完成了所有事情来准备好了一个有效的可调用的就差执行的控制器，但还是有最好的机会来彻底的用一些其他的控制器（其可以是任何可调用的）来替换它。这个机会是由另外一个事件提供的：KernelEvents::CONTROLLER (kernel.controller)：



通过调用 FilterControllerEvent 对象的 setController() 方法，可以覆盖应该被执行的控制器：



**事件传递**

当你覆盖一个中间结果，比如当你在监听到 kernel.filter\_controller 事件后完全替代一个控制器，你或许想要保护其他做同样事情的监听器。你可以通过调用下边的代码来做到：



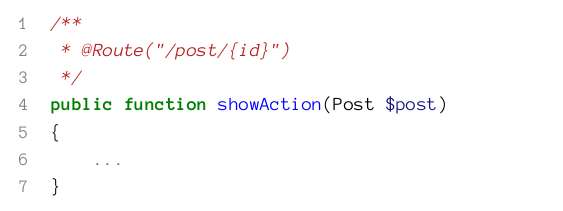
并且确保你的监听器有较高的优先级，好被首先调用。可以看看登记事件监听器。

**一些知名的 kernel.controller 监听器**

这个框架本身没为 kernel.controller 事件提供监听器。只有监听此事件以预知控制器将被执行的第三方 bundle 。

例如来自 SensioFrameworkExtraBundle 的 ControllerListener 在刚刚执行一个控制器后做了一些非常重要的工作：它收集类似 @Template 和 @Cache 之类的 annotation，并把它们以同名加下划线前缀的形式作为请求属性存储起来：\_template 和 \_cache 。稍后在这个请求的处理过程中，这些 annotation（或在这个 bundle 的代码里被调用的配置）将被用来渲染一个模板，或设定一些涉及缓存的响应头。

来自这个 bundle 的 ParamConverterListener 将转换额外的控制器参数，比如通过从路由获取 id 值待处理的实体：



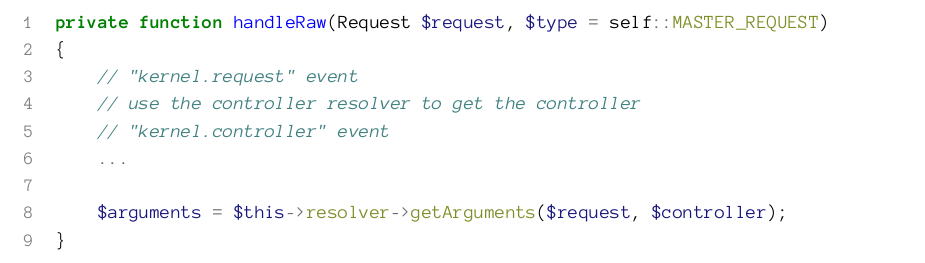


**参数转换器**

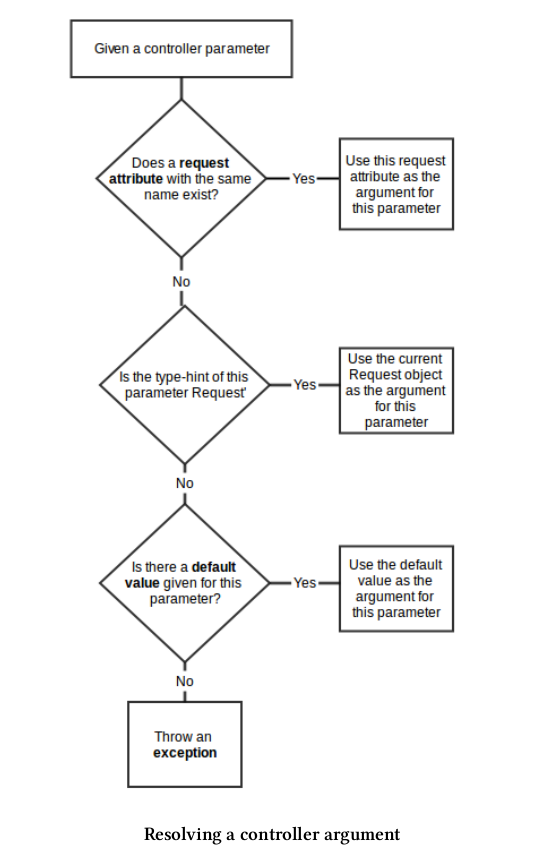
SensioFrameExtraBundle 与 DoctrineParamConverter 交互来帮助字段的名称/值的组合转换成一个实体或文件存储。但你也可以自己创建这些“参数转换器”。只要实现 ParamConverterInterface, 为它创建一个服务定义，赋予它一个 request.param\_converter 标签。 也可以查看文档 [@ParamConverter](http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html)

**2.4 收集用于执行控制器的参数**

在任何用于替代控制器的监听器被开启后，我们现在拥有的控制器就是不可更改的了。下一步是收集可以用于执行这个控制器的参数：

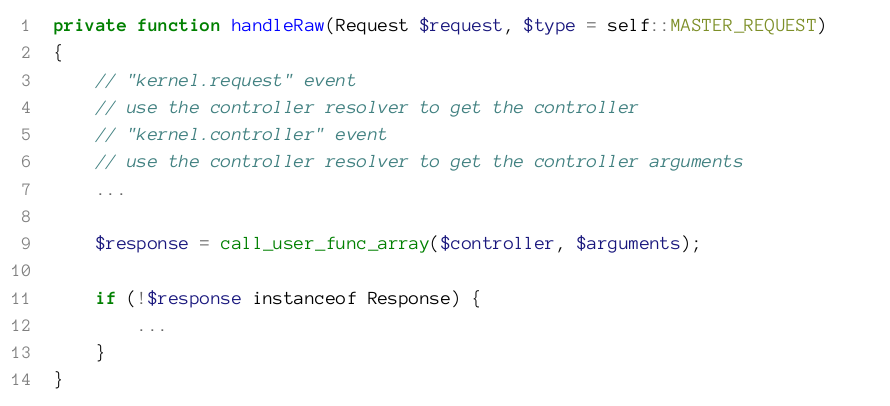


这控制器解析器被要求提供控制器参数。来自 HttpKernel 组件的标准的 ControllerResolver 使用了反射和Request 对象的属性来解析控制器参数。它遍历来自控制器方法的全部参数。下边的逻辑被用来判断每个参数：



**2.5 执行这个控制器**

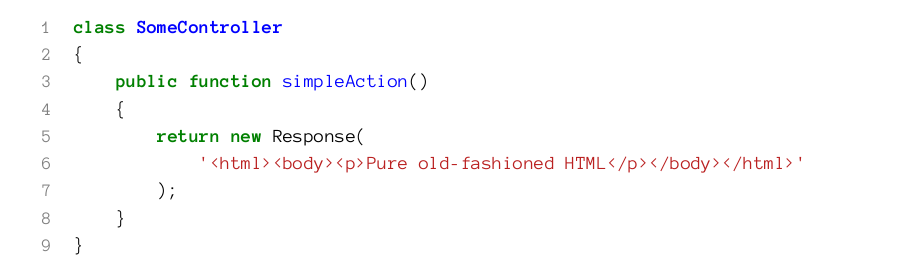
最终，是时候执行这个控制器了。响应被触发并深入处理。



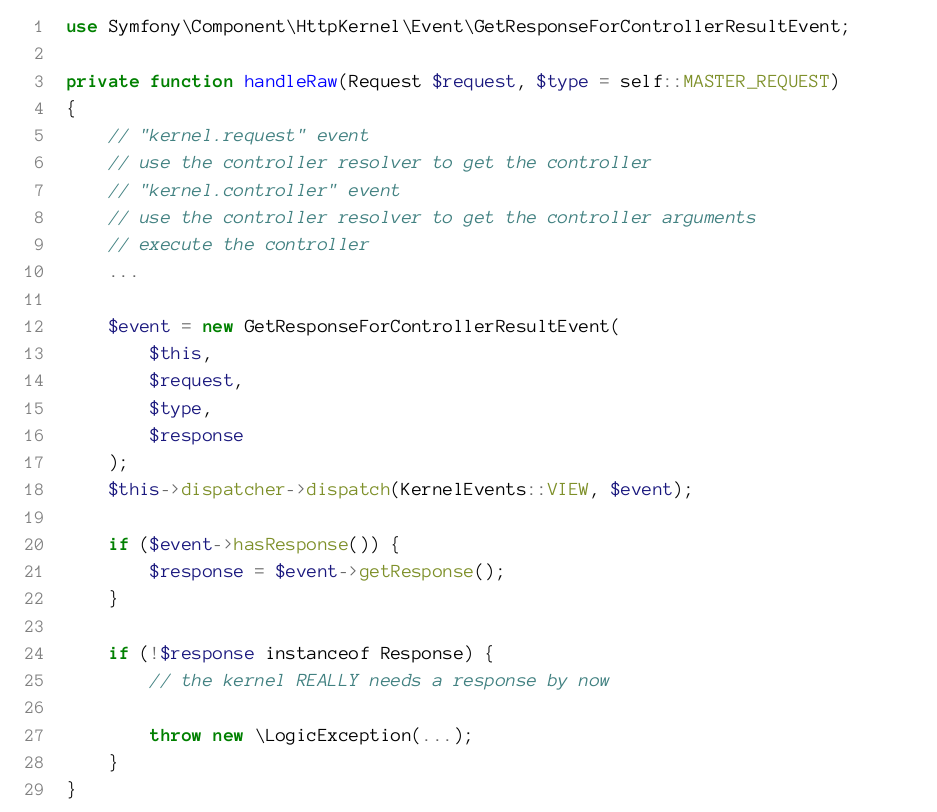
或许你还记得 Symfony 文档所述，一个控制器应该返回一个 Response 对象。如果没返回这个对象，此程序的其他部分应该以某种方式将返回的值转化成一个 Response 对象。

**2.6 进入视图层**

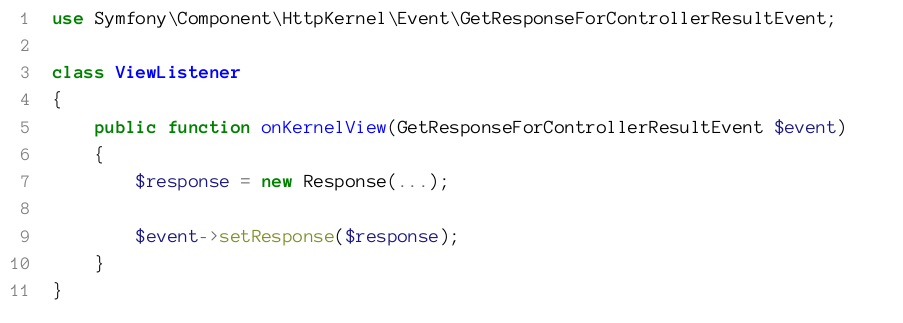
当你选择直接从你的控制器直接返回一个 Response 对象，你可以由此让模板引擎生效：



然而，当你返回任何别的（尽管通常是一个模板变量数组），在它被用作一个得体的响应被发送回给客户端前，这个返回值需要被转化成一个 Response 对象。HttpKernel 并没明确地匹配一个特定的如 Twig这样的模板引擎。取而代之再次使用事件调度来允许任何对 KernelEvents::VIEW 事件的监听器来基于控制器的返回值设定一个适合的响应（尽管它可能选择直接忽略这个值）：

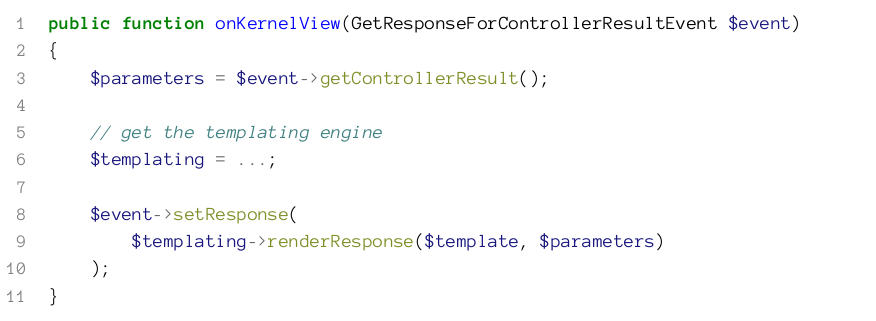


这个事件的监听器可以在 GetResponseForControllerResultEvent 对象上调用 setResponse() ：



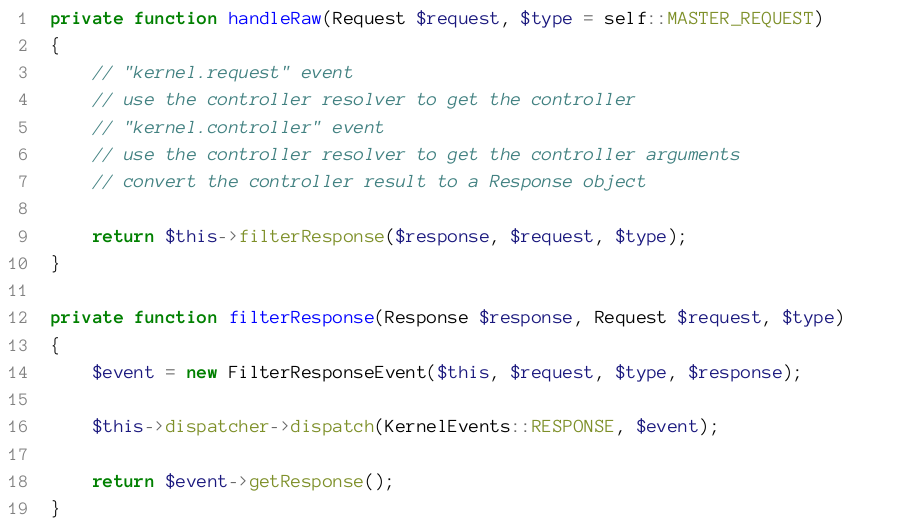
**一个知名的 kernel.view 监听器**

这个来自 SensioFrameworkExtraBundle 的 TemplateListener 将控制结果作为模板变量来渲染那些由 @Template annotation 设定的模板（其存储在请求属性 \_template 里）：



**2.7 过滤响应**

最后，在返回 Response 对象作为处理给定 Request对象的最终结果之前，任何针对 KernelEvents::RESPONSE 事件（kernel.reponse）的监听器都将被调用：



事件监听器允许修改 Response 对象甚至完全替换它：



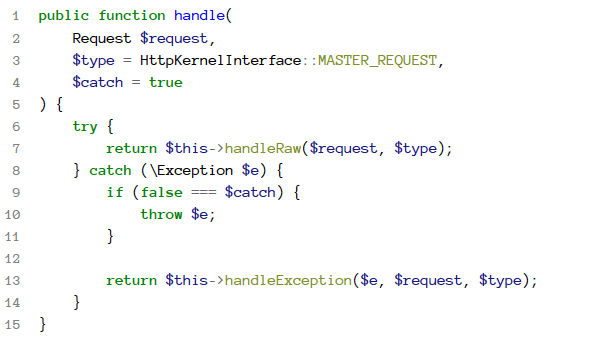
**知名的 kernel.response 监听器**

来自 WebProfilerBundle 的 WebDebugToolListener 在响应的最终阶段注射一些 HTML 和 JavaScript 代码来确保 profiler 工具栏可见（通常在每个页面的底部）。

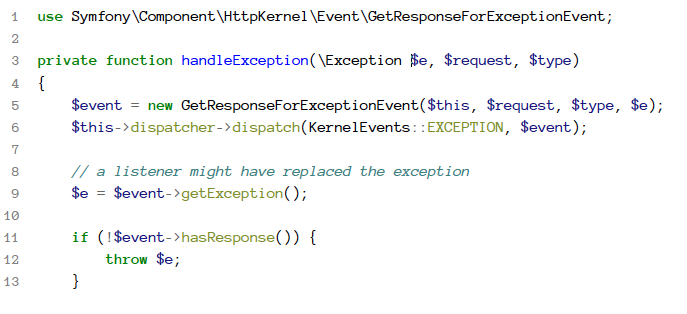
来自Symfony安全组件的ContextListener在session中存储了一个当前安全令牌的序列化版本。这样做能够在下一次请求是快速进行验证过程。安全组件也有一个ResponseListener用来设置cookie包含“记住我”的信息。它的内容可以用于自动登录用户，即使他的原始会话已被销毁。

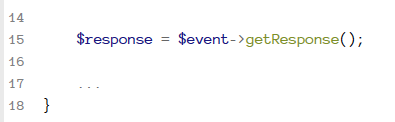
**3 异常处理**

从请求到响应的长途旅程中很可能会出现一些错误或异常。默认情况下，内核被指定捕捉所有异常和事件，然后会找到一个合适的它的响应。正如我们已经看到的那样，整个请求处理被包装进来了一个try / catch块：



当 $catch 是 true 时，handleException() 方法被调用并预计会创建一个响应。这个方法会以一个 GetResponseForExceptionEvent 对象派遣一个 KernelEvents::EXCEPTION 事件(kernel.exception)。



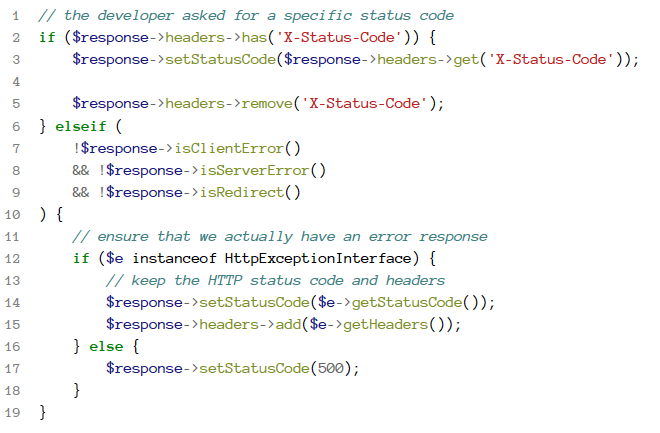


kernel.exception 事件的兼容器被允许：

* 为这个特定的异常设置一个合适的 Response 对象。
* 替换原有的 Exception 对象。

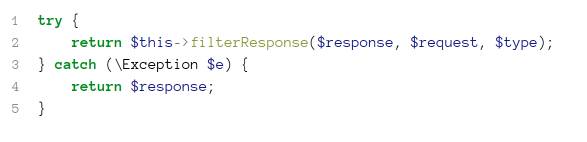
如果没有一个监听器在这个event 对象上调用 setResponse()，这个异常将(再次)被抛出，但这次它不会再被自动处理了。因此一旦你的 PHP 设定的 display\_errors 等于 true，PHP只是呈现它。

一旦有任何一个监听器设定一个 Response 对象，HttpKernel 会检查这个对象，这样可以为这个响应设定一个正确的状态码：



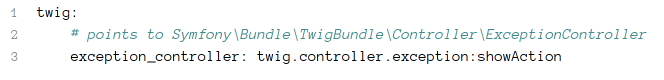
这十分有用：我们可以通过往 Response 对象添加一个 X-Status-Code 头信息来强制使用一个特定的状态码（记住：这个只能针对通过 HttpKernel 不过的异常来工作），或者针对抛出的完成了 HttpExceptionInterface 的异常。否则状态码会默认为500——内部无服务器错误。这比标准的PHP行为要好的多，它会返回一个状态200的响应——是的，当出现一个错误时。

当一个事件监听器设定一个 Response 对象时，这个响应不会受到与其他普通响应不同的对待，所以处理异常的最后一步是过滤这个响应。当在过滤此响应时另一个异常被抛出了，这个异常会被简单的忽略掉，而没被过滤的响应会发送到客户端。



**3.1 知名的 kernel.exception 监听器**

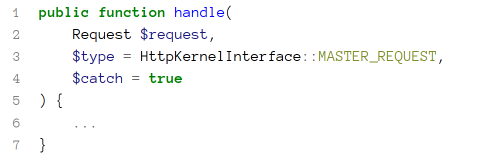
来自 HttpKernel 组件的 ExceptionListerner 自身会尝试通过记录日志来处理一个异常（当 logger 可得时）并执行一个控制器渲染一个页面来显示关于错误的一些信息。通常就是定义在 config.yml 的这个控制器：



另一个重要监听器是来自安全组件的 ExceptionListener 。这个监听器会检查抛出的原始的异常是否是一个 AuthenticationException 或 AccessDeniedException 的一个实例。第一种情况，如果可能它会开始验证过程。第二种情况，它会尝试重新验证此用户或者让准入拒绝处理器来处理这种情况。

**4 子请求**

或许你注意到了 HttpKernel::handle() 方法的参数 $type：



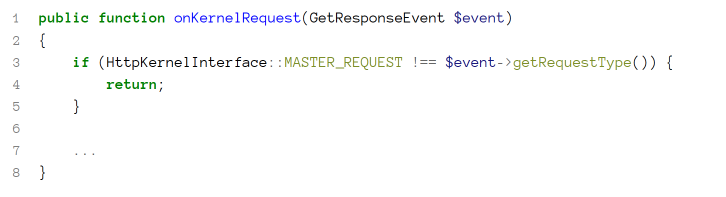
有两个请求类型作为常量定义在 HttpKernelInterface 里：

1. HttpKernelInterface::MASTER\_REQUEST，这是主要请求
2. HttpKernelInterface::SUB\_REQUEST, 一个子请求

每个到你PHP程序的请求，第一个由内核处理的请求是第一种类型的，HttpKernelInterface::MASTER\_REQUEST。这是固定的，因为它是通过在前部控制器放置 $type 参数来起效的（app.php 或 app\_dev.php）:



上边讨论的许多监听内核事件的事件监听器，都仅当请求类型是 HttpKernelInterface::MASTER\_REQEST 时才操作。例如，Firewall 在请求是子请求时不做任何操作。



**4.1 当子请求被使用时？**

子请求用来单独的创建一个 Response 对象。例如，当内核捕获一个异常，标准的异常处理器尝试执行一个设计好的异常控制器（如上）。要做到这点，需要创建一个子请求：



以及，当你将另一个控制作为一个Twig模板的一部分渲染时，也会创建并当做一个子请求来处理：



**当写你自己的内核事件监听器时…**

问一下你自己这个事件监听器是否应该作用在主请求，子请求或两者都要。使用一个guard子句来提早返回（看上边的例子）。

**II 依赖注入的模式**

**5 什么是Bundle？**

如我们前面章节所见：运行一个Symfony 程序意味着启动内核并处理一个请求或执行一个命令，其中启动内核意味着：加载所有的bundle并登记它们的服务容器扩展（其可以在bundle 的DependencyInjection目录找到）。容器扩展通过加载一个 services.xml 文件（但可以是任何文件）以及此 bundle 的配置，其定义在分开的类里，通常在相同的命名空间下，叫 Configuration 的。这些东西合在一起（bundle, 容器扩展和配置）可以用来连接你的bundle：你可以定义参数和服务，这样你提供给你bundle的功能就可以让程序的其他部分也获取到了。你甚至可以更深入一步，注册编译器参数在它形成最终形式之前来深入修改服务容器。

在创建了许多bundle之后，我总结了作为一个 Symfony 程序的开发人员我的工作包括为一下内容编写代码：bundle, 扩展和配置类和编译器传入参数。当你知道怎样编写好的代码时，你仍需要了解怎么创建好的bundle，这基本意味着你需要了解怎样创建好的服务定义。有很多方法可以达到这个目的，我会在本章介绍其中的大部分。了解了你的选项有助于当寻找一个依赖注入模式时做出好的抉择。

**不要使用生成命令**

当你开始使用Symfony时，你可能会试图使用SesioGeneratorBundle提供的命令来生成 bundle，控制器，数据实体和表单类型。我并不建议这么做。这些生成的类可以作为手动创建这些类的很好的范本，但通常它们包含了太多你不需要的代码，或者那些你起始阶段并不需要的东西。所以就用这些命令一次就好，看看应该怎么做，然后自己学着去做，而不是一直依赖这些命令。这会让你成为更好的理解这个框架的更快的开发者。

6 服务模式

一个服务是一个对象，在服务容器里以一个特定的id注册。