第 12 章　缓存

有一种提高应用程序速度的方法—存储生成的 HTML 代码片段，或者整个页面，以后直接使用存储的内容。这种技术叫做缓存，它可以在服务器端和客户端实现。

symfony 提供了一个灵活的服务器端缓存系统。通过很直观的 YAML 文件设置，它可以保存整个回应、一个动作、一个局部模板或者一个模板片段的结果到一个文件里。 当对应的数据变化时，你可以很方便的使用命令行或者特殊的动作方法有选择的清除缓存。symfony 还提供了一个通过 HTTP 1.1 头信息控制客户端缓存的简单方法。本章将详细介绍这些内容，并且还有一些监控缓存对程序性能影响的技巧。

缓存回应

HTML 缓存的原理很简单：重用之前类似请求的部分或者全部 HTML 代码。这些 HTML 代码存储在一个特定的地方（symfony 项目的 cache/目录），前端控制器在执行动作之前会先检查这个目录。 如果找到缓存内容，就把它发送到客户端而不执行动作，因此大大加快了执行速度。如果没有缓存内容，就执行动作，然后把动作的结果（视图）保存在 cache/目录供以后使用。

由于所有的页面都可能包含动态内容，所以 HTML 缓存默认是关闭的。网站管理员可以开启它来提高性能。

symfony 能够处理三种不同类型的 HTML 缓存：

- 动作的缓存（包含或者不包含布局）
- 局部模板，组件或者组件槽的缓存
- 模板片段的缓存

前两种类型可以通过 YAML 配置文件来控制。模板片段的缓存是通过模板里的辅助函数来管理的。

全局缓存设置

项目的每个应用程序的不同环境，HTML 缓存机制都可以在 settings.yml 文件里的 cache 部分设置成开启或者关闭（默认）。例 12-1 是一个开启缓存的例子。

例 12-1 - 开启缓存， myapp/config/settings.yml

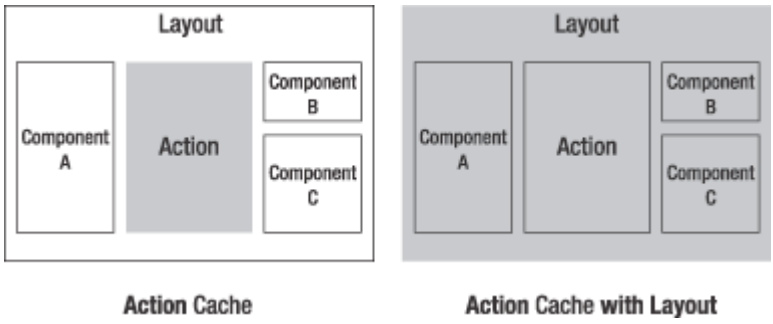
```
dev:
  .settings:
```

```
  cache:                  on
```

缓存一个动作

显示静态内容的动作（不依赖数据库或者与 session 无关的数据）或者从数据读取信息的动作（比如，GET 请求），这类的动作通常比较适合作缓存。图 12-1 显示了不同情况下页面的哪些部分被缓存：动作结果（它的模板）或者动作结果与布局一起。

图 12-1 - 缓存动作



例如，有一个 user/list 动作，它返回网站所有用户的列表。除非有用户被修改、增加或者删除（这种情况会在“从缓存里移除内容”这一小节里讨论），这个动作都会显示同样的内容，所以它非常适合作缓存。

可以在 config/目录的 cache.yml 文件里设置各个动作的开启关闭。请看例 12-2 里的例子。

例 12-2 - 为一个动作开启缓存，myapp/modules/user/config/cache.yml

```
list:
  enabled:     on
  with_layout: false   # 默认值
  lifetime:    86400   # 默认值
```

这个配置里开启了 list 动作的缓存，布局不会与动作一起缓存（一起缓存是默认设置）。这就是说，即使这个动作已经被缓存了，布局（还有其中的局部模板与组件）还是会被执行。如果 with_layout 设置成 true，那么布局就会与动作一起被缓存而不会再次执行。

测试缓存的设置，在你的浏览器里执行测试环境的这个动作。

```
http://myapp.example.com/myapp_dev.php/user/list
```

你会注意到页面的动作区域的边框，第一次，这个区域有一个蓝色的头部，这说明它不是缓存的内容，刷新页面，动作区域有一个黄色的头部，这说明这是缓存的内容（并且速度提升明显）。在本章你会了解更多测试与检测缓存的方

法。

NOTE 槽是模板的一部分，缓存动作的同时也会保存这个动作的模板里定义的槽的值。所以槽可以被缓存。

缓存系统也可以对有参数的页面起作用。假设 user 模块有一个 show 动作，它根据参数 id 显示一个用户的资料。修改 cache.yml 文件这个动作的缓存也打开，如例 12-3 所示。

为了更好地组织 cache.yml 文件，可以把一个模块的所有动作的设置放在 all: 键下，如例 12-3 里所示。

例 12-3 - 完整的 cache.yml 文件示例，
myapp/modules/user/config/cache.yml

```
list:
  enabled:    on
show:
  enabled:    on

all:
  with_layout: false   # 默认值
  lifetime:    86400   # 默认值
```

现在，每次用不同的 id 参数执行 user/show 动作会在缓存里新增一条记录。所以下面这个的缓存：

http://myapp.example.com/user/show/id/12

与下面的缓存不一样：

http://myapp.example.com/user/show/id/25

CAUTION 通过 POST 方法调用的动作或者通过直接的 GET 参数（不通过 symfony 的路由系统转义，直接指定 GET 参数）调用的动作不会被缓存。

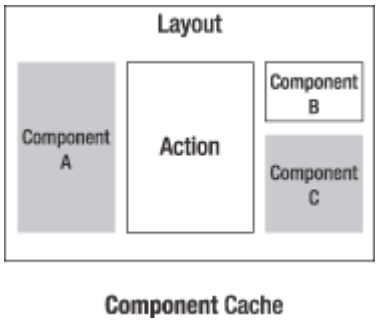
with_layout 这个设置还需要多作一点说明。这个参数决定了缓存里存放哪种数据。不缓存布局的情况下，只有模板的执行结果与动作变量存放在缓存里。缓存布局的时候，整个回应对象都会缓存。这就是说缓存布局要比不缓存布局要快很多。

如果功能上允许（也就是说，布局不依赖于与 session 有关的数据），你应该选择缓存布局。不幸的是，布局经常会包含动态元素（例如，登录的用户的名字），所以缓存动作的时候不包括布局是最常见的配置。不过，RSS 种子，弹出窗口，还有不依赖于 cookies 的页面可以连同布局一起缓存。

缓存一个局部模板，组件或者组件槽

第 7 章介绍了如何在多个模板里使用 include_partial()辅助函数重用代码片段。局部模板与动作一样非常容易缓存，局部模板缓存开启的规则也一样，如图 12-2 所示。

图 12-2 - 缓存一个局部模板，组件或者组件槽



例如，例 12-4 展示了如何通过编辑 cache.yml 文件来开启 user 模块的 _my_partial.php 这个局部模板的缓存。注意 with_layout 设置在这里是没有意义的。

例 12-4 - 缓存一个局部模板，myapp/modules/user/config/cache.yml

```
_my_partial:
  enabled:    on
list:
  enabled:    on
...
```

现在所有使用到这个局部模板的模板都不会真正执行这个局部模板的 PHP 代码，而是直接使用缓存里的内容。

```
<?php include_partial('user/my_partial') ?>
```

与动作类似，局部模板的结果也会跟参数有关。缓存系统把所有不同参数的局部模板的结果保存下来。

```
<?php include_partial('user/my_other_partial', array('foo' => 'bar'))
?>
```

TIP 动作缓存要比局部模板缓存功能强大，因为当动作被缓存时，模板是不会执行的；如果模板包含局部模板的调用，这些调用也不会执行。所以，局部模板缓存只在不使用动作缓存或者布局里的局部模板时有用。

让我们回顾一下第 7 章：组件是局部模板之上的轻量级的动作，组件槽是一个

动作会随着调用动作而变化的组件。这两种包含类型很类似于局部模板，缓存的方法是一样。例如，如果你的全局布局用 include_component('general/day')包含 day 这个组件，用来显示当前日期，设置 general 模块的 cache.yml 文件为如下内容可以开启这个组件的缓存：

```
_day:
  enabled: on
```

缓存组件或者局部模板的时候，你要决定是为所有调用的模板保存单一的版本还是为每个调用的模板保存一个版本。默认情况，缓存的组件与调用它的模板是无关的。不过与环境有关的组件，例如一个在不同的动作里显示不同的侧边栏的组件，应该为每一个调用它的模板保存一个缓存版本。缓存系统可以处理这种情况，只要把 contextual 参数设置成 true，如下：

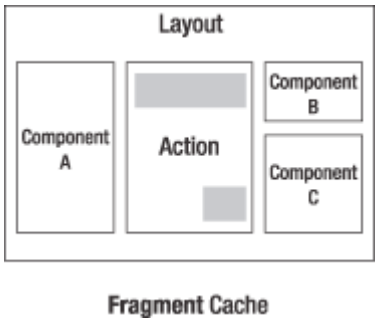
```
_day:
  contextual: true
  enabled:    on
```

NOTE 全局组件（位于应用程序的 template/目录）也可以缓存，这需要在应用程序的 cache.yml 里声明它的缓存设置。

缓存模板片段

动作缓存只适合一部分动作。但是对于其他的动作（用来更新数据或者在模板里显示与 session 有关的信息的动作）仍然有进行缓存提升性能的可能，不过要使用不同的方法。symfony 还提供了第三种缓存，专门用来缓存模板片段并且可以直接在模板里使用。在这种模式下，动作总是会执行，模板被分成执行片段和缓存片段，如图 12-3 所示。

图 12-3 - 缓存模板片段



例如，一个用户列表里有一个到最近访问用户的链接，这个内容是动态的。cache()辅助函数定义模板的那些部分的内容要放到缓存里。语法详情见例 12-5。

例 12-5 - 使用 cache()辅助函数，myapp/modules/user/templates/listSuccess.php

```
<!-- 每次都执行的代码 -->
<?php echo link_to('last accessed user', 'user/show?id='.
$last_accessed_user_id) ?>

<!-- 缓存代码 -->
<?php if (!cache('users')): ?>
  <?php foreach ($users as $user): ?>
    <?php echo $user->getName() ?>
  <?php endforeach; ?>
  <?php cache_save() ?>
<?php endif; ?>
```

它是这样工作的：

- 如果找到了名为 users 的缓存片段， 就用它代替<?php if (!cache($unique_fragment_name)): ?> 与 <?php endif; ?>之间的代码。
- 如果没有，则执行这俩行之间的代码然后保存到缓存，命名为 users。

不在这两行之间的代码总会执行而不会被缓存。

CAUTION 这个动作（例子中的 list）的缓存必须是关闭的，因为开启缓存以后会忽略整个模板还有模板片段的声明。

使用模板片段缓存对速度的提升没有动作缓存的明显，因为动作还是会执行，模板也要处理一部分，布局也还是会用来装饰。

你可以在同一个模板里声明多个模板片段缓存；不过，你必须给它们起不同的名字好让 symfony 缓存系统找到它们。

与动作和组件缓存一样，模板片段缓存的 cache()辅助函数也可以接受持续时间的秒数作为第二个参数。

```
<?php if (!cache('users', 43200)): ?>
```

调用这个辅助函数时如果不指定，则使用默认缓存持续时间（86400 秒或 1 天）。

TIP 还有一个办法可以使这种的动作变得可以缓存，那就是在动作的路由模式（路径）中加入变量。例如，如果首页显示登录的用户名，那么只要使这个页面的 URL 包含用户名这个动作就可以缓存了。另一个例子是对于国际化的项目：如果你想缓存一个有多种语言的页面，语言代码必须包含在 URL 的某处。这个方法增加了缓存里存放的页面数，不过它可以大大加快很繁忙的交互应用程序的速度。

动态配置缓存

cache.yml 是一种定义缓存设置的方法，但是要修改 cache.yml 的设置比较麻烦。不过，与 symfony 其他的地方一样，你可以用 PHP 代替 YAML，这样就可以动态设置缓存了。

为什么需要动态设置缓存呢？有个例子可以很好的说明这个问题，有一个页面登录用户和未登录用户看到的这个页面是不一样的，但是 URL 相同。假设 article/show 页面包含一个文章评分系统。未登录用户不能使用评分系统。这些未登录用户看到的是一个登录表单而不是评分系统。 这个版本的页面可以被缓存。另外一方面，登录用户点击一个评分链接会发出一个 POST 请求并做出评分。这次，这个页面不需要缓存而应该是动态建立。

动态缓存应该在 sfCacheFilter 执行之前的过滤器(filter)里设置。事实上，缓存是 symfony 里的一个过滤器，这与网页调试工具条和安全功能一样。如果要设置 article/show 的缓存只针对未登录用户开启，需要在应用程序的 lib/目录里加一个 conditionalCacheFilter，内容见例 12-6。

例 12-6 - 通过 PHP 配置缓存，
myapp/lib/conditionalCacheFilter.class.php

```
class conditionalCacheFilter extends sfFilter
{
  public function execute($filterChain)
  {
    $context = $this->getContext();
    if (!$context->getUser()->isAuthenticated())
    {
      foreach ($this->getParameter('pages') as $page)
      {
        $context->getViewCacheManager()->addCache($page['module'],
$page['action'],array('lifeTime' => 86400));
      }
    }

    // Execute next filter
    $filterChain->execute();
  }
}
```

你还要在 filter.yml 文件里 sfCacheFilter 的地方注册这个过滤器，如例 12-7 。

例 12-7 - 注册自己的过滤器，myapp/config/filters.yml

```
...
security: ~

conditionalCache:
  class: conditionalCacheFilter
  param:
    pages:
      - { module: article, action: show }

cache: ~
...
```

清除缓存（为了自动载入新的过滤器类），条件缓存就可以用了。它会使 pages 参数指定的页面的缓存在用户未登录时生效。

sfViewCacheManager 对象的 addCache()方法需要一个模块名，动作名，还有一个与 cache.yml 文件里指定的参数内容一样的关联数组作为参数。例如，如果你想定义 article/show 动作必须与布局一起缓存，时间限制是 3600 秒，那么这样写代码：

```
$context->getViewCacheManager()->addCache('article', 'show', array(
  'withLayout' => true,
  'lifeTime'   => 3600,
));
```

SIDEBAR 其他的缓存存储方式

默认情况，symfony 缓存系统将数据保存在服务器的硬盘上。你可能会想把缓存放在内存里（例如，通过 memcache）或者数据库里（特别是你想在几台服务器之间共享缓存数据或者加快缓存删除速度）。你可以轻松的修改 symfony 默认的缓存存储系统，因为 symfony 视图缓存管理器使用的类定义在 factories.yml 文件里。

默认的视图缓存存储 factory 是 sfFileCache 类：

```
view_cache:
    class: sfFileCache
    param:
      automaticCleaningFactor: 0
      cacheDir:                %SF_TEMPLATE_CACHE_DIR%
```

你可以用自己的存储类代替这个类或者使用 symfony 提供的其他的类（例如 sfSQLiteCache）。param 键下定义的参数会作为关联数组传给这个类的 initialize()方法。所有的视图缓存存储类都必须定义抽象类 sfCache 里定义

的方法。详情请参考 API 文档（http://www. symfony-project.com/api/symfony.html）。

使用极速缓存

即使是缓存的页面也需要执行一些 PHP 代码。缓存的页面，symfony 还是要载入配置文件，建立回应等。如果你非常确定一个页面在一段时间内不会改变，你可以完全绕过 symfony，直接把生成的 HTML 代码放在 web/目录下。这需要 Apache 的 mod_rewrite，还有你的路由规则需要指定没有后缀或者以.html 为后缀。

你可以手动的，一页一页的，使用这个简单的命令作这件事：

```
> curl http://myapp.example.com/user/list.html > web/user/list.html
```

设置好以后，每次请求 user/list 动作的时候，Apache 会找到对应的 list.html 文件然后完全忽略 symfony。这样做的代价是你不能用 symfony 控制页面缓存了（生存时间，自动删除等），但是速度提高非常明显。

另外，你可以使用 sfSuperCache 这个 symfony 插件，它能自动生成极速缓存并且支持存活时间和清除缓存。插件的使用请参考第 17 章。

SIDEBAR 其他的加速策略

除了 HTML 缓存之外，symfony 还有两种其他的缓存机制，这两种缓存机制是完全自动并且对开发者透明的。在生产环境，配置和模板翻译的缓存会自动放在 myproject/cache/config/ 和 myproject/cache/i18n/目录里。

PHP 加速器（eAccelerator，APC，XCache 等），也被称作 opcode 缓存模块，可以通过缓存 PHP 代码编译后的状态来提高 PHP 代码的执行速度，所以代码解析还有编译的负担可以完全被消除。这对包含大量代码的 Propel 类特别有用。这些加速器与 symfony 兼容并且可以轻易的将程序的速度提高三倍。所以推荐流量大的 symfony 应用程序在生产环境使用这些加速器。

使用 PHP 加速器，你可以使用 sfProcessCache 类在内存里保存持久数据，这样可以避免每次请求作同样的处理。另外如果你想保存一个很消耗 CPU 的函数的结果到一个文件，你可以使用 sfFunctionCache 对象。关于这些机制的详细信息，请参考第 18 章。

从缓存里删除项目

如果程序的脚本或者数据发生了改变，那么缓存的数据会过期。为了避免这样的情况同样也为了避免 bug，你需要根据需要用不同的方法删除缓存的部分内容。

删除整个缓存

symfony 命令行的 clear-cache 任务可以删除缓存（HTML，配置文件还有 i18n 的缓存）。你可以指定参数让它只删除一部分缓存，如例 12-8 所示。请注意只能在 symfony 项目的根目录执行这个命令。

例 12-8 - 清除缓存

```
// 清除整个缓存
> symfony clear-cache

// 简写
> symfony cc

// 只清除 myapp 应用程序的缓存
> symfony clear-cache myapp

// 只清除 myapp 应用程序的 HTML 缓存
> symfony clear-cache myapp template

// 只清除 myapp 应用程序的配置缓存
> symfony clear-cache myapp config
```

清除指定的缓存

数据库更新以后，与修改的数据有关动作的缓存必须清除。你可以清除整个缓存，不过这就把其他不相关的缓存内容浪费了。这里就需要 sfViewCacheManager 对象的 remove()方法。它需要一个内部 URL 作为参数（与传给 link_to()的一样），它会删除相关的动作缓存。

例如，假设 user 模块的 update 动作修改 User 对象的字段。那么 list 和 show 动作的缓存就需要清除，否则包含错误数据的旧版本就会显示出来。可以使用 remove()方法来处理，如例 12-9 所示。

例 12-9 - 清除指定动作的缓存， modules/user/actions/actions.class.php

```
public function executeUpdate()
{
  // 更新用户
  $user_id = $this->getRequestParameter('id');
  $user = UserPeer::retrieveByPk($user_id);
  $this->foward404Unless($user);
  $user->setName($this->getRequestParameter('name'));
  ...
```

```
  $user->save();

  // 清除与这个用户有关的动作的缓存
  $cacheManager = $this->getContext()->getViewCacheManager();
  $cacheManager->remove('user/list');
  $cacheManager->remove('user/show?id='.$user_id);
  ...
}
```

删除缓存的局部模板，组件或者组件槽有点复杂。你可以传给它们任何种类的参数（包括对象），这之后你几乎无法识别缓存的版本。这里我们重点介绍局部模板，这与其他的模板缓存是相同的。symfony 用一个特殊的前缀（sf_cache_partial）标识一个缓存的局部模板，然后再加上模块的名字、局部模板的名字还有所有参数的哈希值，如下所示：

```
// 调用一个局部模板
<?php include_partial('user/my_partial', array('user' => $user) ?>

// 这个局部模板在缓存中的标识
/
sf_cache_partial/user/_my_partial/sf_cache_key/bf41dd9c84d59f3574a5da
244626dcc8
```

理论上，如果你知道识别参数的哈希值你可以用 remove()方法删除一个缓存的局部模板，但这是很不现实的。幸运的是，如果你在 include_partial()辅助函数调用的之后指定一个 sf_cache_key 参数，你就可以用你知道的东西来标识这个缓存。 例 12-10 里，清除一个缓存的局部模板，清除修改过的 User 相关的局部模板，变得很容易。

例 12-10 - 清除缓存里的局部模板

```
<?php include_partial('user/my_partial', array(
  'user'          => $user,
  'sf_cache_key' => $user->getId()
) ?>

// 这个模板在缓存里的标识
/sf_cache_partial/user/_my_partial/sf_cache_key/12

// 清除某个特定的 user 的_my_partial，使用 $cacheManager-
>remove('@sf_cache_partial?
module=user&action=_my_partial&sf_cache_key='.$user->getId());
```

使用这个方法并不能清除一个局部模板的所有缓存。这些你会在本章后面的“手动清除缓存”里了解到。

要清除模板片段的缓存，也可以用同样的 remove()方法。缓存里模板片段的标识符由跟刚才一样的 sf_cache_partial 前缀，模块名，动作名还有 sf_cache_key（使用 cache()辅助函数使指定的不重复的名字）组成。见例 12-11。

例 12-11 - 清除模板片段的缓存

```
<!-- 缓存的代码 -->
<?php if (!cache('users')): ?>
  ... // Whatever
  <?php cache_save() ?>
<?php endif; ?>

// 这个缓存的标识符使用了缓存页面的网页调试工具条
/sf_cache_partial/user/list/sf_cache_key/users

// 这样清除它
$cacheManager->remove('@sf_cache_partial?
module=user&action=list&sf_cache_key=users');
```

SIDEBAR 选择性清除缓存比较伤脑筋

缓存清除工作中最麻烦的事情是判断那些动作受到数据更新的影响。

例如，假设当前应用程序有一个 publication 模块，这个模块显示出版物列表（list 动作）和连同作者（User 类的实例）详细信息的出版物描述（show 动作）。修改一条作者记录会影响这个作者所有的出版物列表及其详细描述。这意味着你需要在 user 模块的 update 动作里增加一些这样的东西：

```
$c = new Criteria();
$c->add(PublicationPeer::AUTHOR_ID, $this-
>getRequestParameter('id'));
$publications = PublicationPeer::doSelect($c);

$cacheManager = sfContext::getInstance()->getViewCacheManager();
foreach ($publications as $publication)
{
  $cacheManager->remove('publication/show?id='.$publication-
>getId());
}
```

```
$cacheManager->remove('publication/list');
```

当你开始使用HTML缓存的时候，你要清楚地了解模型和动作之间的相关性，这样才不会出现因为错误的关系造成的新问题。注意如果应用程序里使用了HTML缓存所有的修改模型的动作都应该包含一些remove()方法的调用。

不过，如果你不想伤脑筋来去作复杂分析，完全可以每次更新数据的时候清除整个缓存。

缓存目录结构

应用程序的cache/目录包含下面的结果：

```
cache/                  # sf_root_cache_dir
  [APP_NAME]/           # sf_base_cache_dir
    [ENV_NAME]/         # sf_cache_dir
      config/           # sf_config_cache_dir
      i18n/             # sf_i18n_cache_dir
      modules/          # sf_module_cache_dir
      template/         # sf_template_cache_dir
        [HOST_NAME]/
          all/
```

缓存的模板存放在[HOST_NAME]（主机名）目录下（为了文件系统的兼容性，点被替换成了下划线），然后按照URL来组织下面的目录结构。例如，下面一个页面的模板缓存：

```
http://www.myapp.com/user/show/id/12
```

存放在：

```
cache/myapp/prod/template/www_myapp_com/all/user/show/id/12.cache
```

在代码里不要直接使用文件路径，应该使用文件路径常量来代替。例如，获取当前应用程序的当前环境下的template/的绝对路径可以使用sfConfig::get('sf_template_cache_dir')。

了解这个目录结构有助于手工清除模板。

手工清除缓存

跨应用程序清除缓存是个麻烦。例如，如果一个管理员通过backend应用程序修改了user表里的一条记录，所有frontend应用程序的与这条数据有关的动作的缓存都需要清除。remove()方法需要的参数是一个内部URL，但是应用程序不知道其他应用程序的路由规则（应用程序间是独立的），所以不能使用

remove()方法来清除其他应用程序的缓存。

解决方法是根据路径手工清除 cache/目录下的文件。例如，如果 backend 应用程序需要清除 frontend 应用程序的 user/show 动作的 id 参数为 12 的缓存，可以使用下面的代码：

```
$sf_root_cache_dir = sfConfig::get('sf_root_cache_dir');
$cache_dir =
$sf_root_cache_dir.'/frontend/prod/template/www_myapp_com/all';
unlink($cache_dir.'/user/show/id/12.cache');
```

但这并不能让人满意。这个命令只能清除当前环境的缓存，而且还需要在文件路径里写明环境名和当前的主机名。使用 sfToolkit::clearGlob()方法可以避免这个问题。它可以接受一个包含通配符的文件路径作为参数。例如，清除上个例子中的缓存文件，不必指定主机名与环境，可以使用下面的代码：

```
$cache_dir = $sf_root_cache_dir.'/frontend/*/template/*/all';
sfToolkit::clearGlob($cache_dir.'/user/show/id/12.cache');
```

这个方法在清除与特定参数无关的动作缓存时也很有用。例如，如果你的应用程序处理多种语言，你会在所有的 URL 里加上语言代码。所以到用户档案页面的 URL 可能会类似这样：

```
http://www.myapp.com/en/user/show/id/12
```

要清除缓存的所有语言的 id 为 12 的用户档案，可以简单的这样做：

```
sfToolkit::clearGlob($cache_dir.'/*/user/show/id/12.cache');
```

缓存测试与监测

HTML 缓存如果处理的不好，可能会造成显示的数据混乱。所以每当你要禁用一个元素的缓存的时候，你都应该完整的测试它并且监测执行速度并进行调整。

建立一个临时工作环境

由于在开发模式下缓存默认是关闭的，所有缓存系统可能造成生产环境中无法察觉的新问题。如果你开启某些动作的 HTML 缓存，你应该增加一个新环境，在本章称作临时工作环境（staging），设置与 prod 环境相同（例如，开启缓存）不过 web_debug 设置成 on。

修改应用程序的 settings.yml，增加例 12-12 里的这几行内容到这个文件的最

前面。

例 12-12 - 设置一个临时工作环境 staging，myapp/config/settings.yml

```
staging:
  .settings:
    web_debug:  on
    cache:      on
```

另外，复制生产环境的前端控制器（比如 myproject/web/index.php）来创建一个新的前端控制器 myapp_staging.php。修改 SF_ENVIRONMENT 和 SF_DEBUG 的值，如下：

```
define('SF_ENVIRONMENT', 'staging');
define('SF_DEBUG',         true);
```

好了，你建立了一个新的环境。在域名后加这个前端控制器的名字来调用：

```
http://myapp.example.com/myapp_staging.php/user/list
```

TIP 除了复制旧的前端控制器，还可以通过 symfony 命令行建立一个新的前端控制器。例如，建立 myapp 应用程序的 staging 环境，文件名为 myapp_staging.php，SF_DEBUG 值为 true，只要使用 symfony init-controller myapp staging true 命令即可。

监测性能

第 16 章会详细介绍网页调试工具条的内容。不过，由于工具条包含了有关缓存元素的有用的信息，下面先简单介绍一下工具条的缓存功能。

浏览包含可缓存元素（动作，局部模板，模板片段等）的页面的时候，网页工具条（在窗口的右上角）上会有一个忽略缓存按钮（绿色的，环行箭头），如图 12-4。这个按钮会从新载入页面并且强制处理缓存的元素。注意它并不会清除缓存。

调试工具条最右边的数字是请求执行的时间。如果对某个页面开启了缓存，第二次打开这个页面的时候这个数字会减少，因为 symfony 使用缓存的数据而不是重新处理脚本。可以很方便的通过这个指示器监视缓存对性能的提升。

图 12-4 - 使用了缓存的页面的网页调试工具条

调试工具条还会显示当前请求执行的数据库查询数，还会按照分类显示本次查

询消耗的时间（点总时间查看详情）。通过监视这些数据及处理总时间，可以帮你衡量缓存带来的性能提高。

基准化分析 Benchmarking

由于要在日志和网页工具条上显示调试信息，调试模式很影响应用程序的速度。所以在临时环境 staging 的处理时间并不能代表生产环境里调试模式关闭状态下的处理时间。

要更好的了解每次请求的时间，需要使用基准化分析工具，例如 Apache Bench 或者 JMeter。这些工具可以进行负载测试并且提供两个重要的信息：某个页面的平均载入时间和服务器的最大处理能力。平均载入时间这个数据在监测缓存带来的性能提升时特别有用。

识别缓存的部分

开启了网页调试工具条以后，页面里缓存的部分会用一个红色的框标识出来，左上角会显示缓存的信息框，如图 12-5 所示。如果这个元素被执行，框会是蓝色背景，如果是黄色背景那么就是缓存的数据。点击缓存信息链接会显示缓存的标识符，它的生存时间和距离上次改变的时间。这在处理没有环境元素的时候比较有用，它可以显示这个元素建立的时间还有模板的哪些部分可以缓存。

图 12-5 - 识别页面里缓存的元素

HTTP 1.1 与客户端缓存

HTTP 1.1 协议定义了一些控制浏览器缓存系统的头信息，这对进一步提高应用程序的速度有很大的帮助。

万维网联盟定义的 HTTP 1.1 规范（W3C, http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html）详细介绍了这些头信息。如果一个动作开启了缓存，并且使用了 with_layout 选项，那么还可以通过下面的一个或者多个方法进一步优化。

即使有些网站访问者的浏览器不支持 HTTP 1.1，使用 HTTP 1.1 的缓存功能也没有任何害处。浏览器接受到不能识别的头信息以后会直接简单的把这样的头信息忽略掉，所以建议使用 HTTP 1.1 缓存。

另外，代理服务器和缓存服务器也能识别 HTTP 1.1 的头信息。即使用户的浏览器不支持 HTTP 1.1，还是有可能通过代理服务器来获得 HTTP 1.1 缓存带来的好处。

增加ETag头信息来避免发送重复的内容

ETag功能开启以后，web服务器会在HTTP头部增加回应信息的签名。

ETag: 1A2Z3E4R5T6Y7U

用户的浏览器会把这个签名保存起来，然后下次请求同一个页面的时候把这个签名一起发出去。如果新的签名比对后发现页面没有改变，那么服务器不会发送回应，而只是发一个304: Not modified的头信息。这样节省了服务器的CPU时间（例如GZIP开启）和带宽（页面传输），还有客户端的时间（页面传输）。这样加起来，有ETag缓存的页面的载入时间比没有ETag缓存的要短。

在symfony里，可以修改settings.yml来为整个应用程序开启 ETag功能。下面是默认的ETag设置：

```
all:
  .settings:
    etag: on
```

缓存了布局的动作，回应是直接从cache/目录里取出的，所以处理速度更快。

增加Last-Modified头信息避免发送仍然有效的内容

服务器向浏览器发送回应的时候，它会增加一个特殊的头部信息来说明数据包含的页面最后修改的时间：

Last-Modified: Sat, 23 Nov 2006 13:27:31 GMT

浏览器能理解这个头信息，当再次请求这个页面的时候，会对应的加上一个If-Modified头信息：

If-Modified-Since: Sat, 23 Nov 2006 13:27:31 GMT

服务器可以比较客户端和应用程序返回的这个值。如果匹配，服务器返回304: Not modified头信息，与ETags很像，这样可以节约带宽和CPU时间。

在symfony里，你可以像其他的头信息一样设定Last-Modified。例如，可以在动作里这么使用：

```
$this->getResponse()->setHttpHeader('Last-Modified', $this-
>getResponse()->getDate($timestamp));
```

这个日期可以是从数据库或者文件系统取得数据更新的真实时间。sfResponse对象的getDate()将时间戳转化成Last-Modified需要的日期时间格式（RFC1123）。

通过增加 Vary 头信息来保存一个页面的多个缓存版本

另一个 HTTP 1.1 头信息是 Vary。它可以定义页面取决于哪个参数，可以被浏览器和代理服务器识别。例如，如果页面的内容取决于 cookies，可以把 Vary 设置成这样：

```
Vary: Cookie
```

多数时候，由于页面内容会随着 cookie、用户语言或者其他因素的影响改变，所以很难开启对动作的缓存。如果你不介意增加缓存的大小，就应该正确地设置回应的 Vary 头信息。可以通过 cache.yml 配置文件或者 sfResponse 相关的方法对整个对象或者其中的某些动作设置 Vary 头信息：

```
$this->getResponse()->addVaryHttpHeader('Cookie');
$this->getResponse()->addVaryHttpHeader('User-Agent');
$this->getResponse()->addVaryHttpHeader('Accept-Language');
```

symfony 会对这些参数的每个值保存一个不同的缓存版本。这会增加缓存的尺寸，不过服务器收到与这些匹配的头信息的时候，回应就直接从缓存里面取而不用处理。这对只取决于请求头信息的页面来说可以大大的提高性能。

通过增加 Cache-Control 头信息来允许客户端缓存

到目前为止，即使增加了缓存有关的 HTTP 头或是存在缓存的页面，浏览器还是会从到服务器请求数据。可以通过增加 Cache-Control 和 Expires 头信息来避免重复请求。这些头信息在 PHP 是默认关闭的，不过 symfony 改写了这个设置来避免不必要的请求。

与之前一样，可以通过调用 sfResponse 对象的方法来开启这个功能。在动作里，定义页面最长的缓存时间（以秒为单位）：

```
$this->getResponse()->addCacheControlHttpHeader('max_age=60');
```

也可以指定页面缓存的条件，防止服务器的缓存里保存私有数据（例如银行帐号）：

```
$this->getResponse()->addCacheControlHttpHeader('private=True');
```

使用 Cache-ControlHTTP 指令，你可以很好的调整服务器和浏览器之间的缓存机制。这些指令的详细信息，请看 W3C Cache-Control 的规范说明。

最后一个可以通过 symfony 设定的头信息是 Expires 头信息：

```
$this->getResponse()->setHttpHeader('Expires', $this->getResponse()->getDate($timestamp));
```

CAUTION 使用了 Cache-Control 之后导致了最主要的一个问题是服务器的日志不会记录所有的请求，它只记录真正收到的请求。如果性能提高了，网站统计信息上的数字反而会减少。

总结

根据不同的缓存类型，缓存系统提供了多种提升性能的方法。效果从最好到最差，缓存分成下面几种类型：

- 极速缓存
- 包含布局的动作缓存
- 不包括布局的动作缓存
- 模板里的片段缓存

另外，局部模板和组件也可以被缓存。

如果改变了模型或者 session 里的数据，需要清除缓存来保持一致性，可以通过微调来优化性能—只清除改变了的东西，保留其他的。

请注意测试所有开启缓存页面的时候要格外小心，如果缓存了错误的数据或者更新数据时忘记更新缓存可能造成新问题。临时工作环境 staging 就是专门为这个准备的。

最后，尽量利用好 HTTP 1.1 协议还有 symfony 的先进缓存调整功能，客户端的缓存可以使性能更进一步提高。