很
en1hi1111g
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111

第 18 章 – 性能

如果你希望你的网站能够吸引很多人，优化网站性能将是在开发阶段的一个主要要素。令人安心的是 symfony 核心开发者总会非常关注性能问题。

通过加速开发带来好处的同时也会带来一些多余的开销，symfony 核心开发者总是会认识到性能的需求。因此，每一个类每一个方法都会仔细的分析并优化到尽可能的快速。基本的开销，可以通过比较使用和不使用 symfony 来显示"hello, world"的时间来测量，这个开销很小。因此，这个框架是可扩展的并能在压力测试下表现的很好。最好的证据是，一些高访问量的网站（有百万活跃用户的并有大量消耗服务器资源的 Ajax 交互的）使用 symfony 并且非常满意它的性能。在 wiki 上可以看一下这用 symfony 开发的网站列表 (http://www.symfony-project.com/trac/wiki/ApplicationsDevelopedWithSymfony)。

不过，很显然高访问量的站点通常会扩展服务器数量并升级到他们想要的硬件。如果你没有足够的资源做到这一点，或者如果你想确保框架的全部力量都在你的掌握中，你可以使用几个调整来进一步加快你的 symfony 应用程序。本章列出了一些在框架所有层次中和更多高级用户的推荐优化性能方法。它们中的一些在以前的章节中已经提过，但是你会觉得把它们都集中在一起会对你十分有帮助。

调整服务器

一个精心优化的应用程序应该放在一个优化良好的服务器上。你应该了解服务器性能的基础知识，以确保 symfony 运行没有瓶颈。这里有几样东西需要查核，以确保你的服务器不会过于缓慢。

在 php.ini 中设置 magic_quotes_gpc 为 on 会降低应用程序效率，因为这会让 PHP 把请求参数中的所有引用都转义，但 symfony 会在后来系统化的过程中还原它们，这样唯一的后果就是时间上的损失--并会在一些平台上带来引用－转义问题。因此，如果能修改 PHP 配置的话，设置这个参数为 off。

PHP 版本越新越好。PHP5.2 比 PHP5.1 快，PHP5.1 比 PHP5.0 快。所以请升级你的 PHP 来获得最新的性能提升。

在生产服务器上使用 PHP 加速器（例如 APC, XCache 或者 eAccelerator）几乎是必须的，因为它能让 PHP 跑的比平均快 50%。安装其中一个加速器扩展来感觉一下 PHP 的真实速度。

此外，在生产服务器上确认关闭了 debug 程序，例如 Xdebug 或者 APD 扩展。

NOTE 你也许会担心 mod_rewrite 扩展的开销：这其实是可以忽略的。确实，通过重写规则来读取一张图片比不通过重写规则来读取慢，但是放慢的量级低于执行任何的 php 语句。

一些 symfony 开发者喜欢使用 syck，这是一个 YAML 分析器，PHP 的一个扩展，它可以替代 symfony 的内部分析器。这确实比较快，但 symfony 的缓存系统已经让 YAML 分析的开销最小化了，所以使用 syck 不会给已有生产环境带来什么益处。你小心 syck 不是很成熟，使用的时候也许会发生错误。不管怎么说，如果你感兴趣，安装这个扩展(http://whytheluckystiff.net/syck/)， symfony 会自动使用它的。

TIP 当一台服务器不够用的时候，你可以增加其他服务器来负载均衡。只要 uploads/目录是共享的并且使用了数据库存储用户会话，symfony 项目会无缝的嵌入负载均衡架构。

调整模型

在 symfony 中，模型层是公认最慢的部分。 如果通过基准程序测试发现需要优化模型层，这里有一些可能的改进方法。

优化 Propel 整合

初始化模型层（核心 Porpel 类）会花一些时间，因为它需要去载入一些类并构造多个对象。无论如何，因为 symfony 整合了 Propel，所以这些初始化任务只会在动作确实需要模型的时候才会发生，并且会尽量晚发生。Propel 类只会在当生成的模型对象自动载入的时候才会被初始化。这就意味不使用模型的页面不会被模型层所累。

如果你的应用程序完全不需要使用模型层，你也能在 settings.yml 中设置关闭所有的层并保存在 sfDatabaseManager 的初始化值中：

```
all:
  .settings:
    use_database: off
```

生成的模型类（在 lib/model/om/）已经被优化过了——他们不包含注释，并且他们从自动载入机制中获益。依靠自动载入替代手动包含文件意味着类会在确实需要的时候才会被载入。因此在不需要模型类的情况下，有自动载入机制会节省执行时间，若使用 include 语法来实现则不会节省时间。 对于注释，他们注解了生成的方法，但是会使模型文件变大——结果会导致轻微的磁盘读取开销。因为生成的方法名是非常清楚的，所以默认注解是关闭的。

这两个加强是针对 symfony 的，但你能通过修改 propel.ini 文件恢复默认值，

如下：

```
propel.builder.addIncludes = true    # 在生成的类中增加 include
                                     # 来替代自动载入机制
propel.builder.addComments = true    # 在生成的类中增加注释
```

限制化合（Hydrate）对象数量

当用 peer 类的方法来获得对象的时候，查询通过化合（hydrating）处理（基于查询结果的行来创建和填充对象）。例如，通常可以使用下面语句通过 Propel 获得 article 表的所有行：

```
$articles = ArticlePeer::doSelect(new Criteria());
```

$articles 变量得到的值是 Article 类的对象数组。每一个对象都被创建并初始化了，这需要一些时间。从而得到一个结论： 相对于直接访问数据库的查询语句，Propel 查询语句的速度直接取决于它返回结果的数量。这就是说你的模型方法应该经过优化过只返回指定数量的结果。当你不需要从 Criteria 获得所有结果的时候，你应该使用 setLimit()和 setOffset()方法来限制返回结果数量。例如，如果你只需要获得第 10-20 行结果，可以如例 18-1 中一样改进一下 Criteria。

例 18-1 - 限制 Criteria 返回的结果数量

```
$c = new Criteria();
$c->setOffset(10);  // 第一个返回记录的偏移量
$c->setLimit(10);   // 返回记录数量
$articles = ArticlePeer::doSelect($c);
```

这可以通过使用翻页来自动完成。sfPropelPager 对象通过自动处理 offset 和 Propel 查询语句的 limit 来获得对象的特定页的数据。更多这个类的信息可以参考 API 文档。

用 Join 让结果数量最小化

在应用程序开发过程中，你应该关注每个请求会产生多少个数据库查询语句。网页调试工具条显示了每页有多少条查询语句，点击数据库图标会显示出这些 SQL 查询语句。如果看到查询语句的数量增加有异常，就要考虑一下使用 join 了。

在解释 join 方法之前，让我们回顾一下循环一个对象数组并用 Propel 获得相关类的资料时会发生什么，如例 18-2 所示。这个例子假设你的设计（schema）描述了一个带有 author 表外键的 article 表。

例 18-2 - 在循环中获得相关类的详细信息

```
// 在动作中
$this->articles = ArticlePeer::doSelect(new Criteria());

// 通过 doSelect()发出的数据库查询
SELECT article.id, article.title, article.author_id, ...
FROM   article

// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
  <li><?php echo $article->getTitle() ?>,
    written by <?php echo $article->getAuthor()->getName() ?></li>
<?php endforeach; ?>
</ul>
```

如果$articles 数组包含了十个对象，那么当类 Author 的对象调用化合（hydrate）的时候会依次执行十次 getAuthor()方法，如例 18-3 所示。

例 18-3 - 外键获取方法发出了一个数据库查询

```
// 在模板中
$article->getAuthor()

// getAuthor()发出的数据库查询
SELECT author.id, author.name, ...
FROM   author
WHERE  author.id = ?                  // ? 是 article.author_id
```

所以例 18-2 中的翻页总共需执行 11 条查询语句： 其中一个当然是用来建立 Article 对象的，其余的 10 条查询语句是用来逐次建立 Author 对象。仅仅显示文章和他们的作者列表却需用多条查询语句来完成。

如果只是使用简单的 SQL 语句，你应该知道如何减少查询语句，只用一条语句来获得 article 表和相关 author 表的内容。这就是 ArticlePeer 类的 doSelectJoinAuthor()方法做的事情。它提供了比单纯 doSelect()调用更复杂的查询语句，但在结果集中增加了列，设置允许 Propel 来融合 Article 对象和相关的 Author 对象。例 18-4 中的代码展示了和例 18-2 同样的效果，但是只需要一条数据库查询语句而不是以前的 11 条语句来处理，这会处理的更快。

例 18-4 - 在一条语句中获得文章详细资料和他们的作者

```
// 在动作中
$this->articles = ArticlePeer::doSelectJoinAuthor(new Criteria());

// doSelectJoinAuthor()发出的数据库查询
SELECT article.id, article.title, article.author_id, ...
       author.id, author.name, ...
FROM   article, author
WHERE  article.author_id = author.id

// 在模板中(没有改变)
<ul>
<?php foreach ($articles as $article): ?>
  <li><?php echo $article->getTitle() ?>,
    written by <?php echo $article->getAuthor()->getName() ?></li>
<?php endforeach; ?>
</ul>
```

调用 doSelect()或 doSelectJoinXXX()方法对返回的结果来说没有区别； 他们都返回了同样的对象数组（如例中的 Article 类）。在这之后使用这些对象的外键获取方法才会看出不同。在用 doSelect()的情况下，他发出了查询, 一个对象会产生一个结果； 而在用 doSelectJoinXXX()的情况下，外部对象已经存在了，不需要用查询语句了，所以处理过程会快些。因此，如果你知道需要用到相关的对象的话，调用 doSelectJoinXXX()方法会减少数据库查询语句的数量——并提高了分页的效率。

doSelectJoinAuthor()方法是根据 article 和 author 表的关系在调用 propel-build-model 时自动产生的。如果在 article 表结构中有其他的外键（例如，对于分类表）生成的 BaseArticlePeer 类就会有其他的 Join 方法，如例 18-5 所示。

例 18-5 - ArticlePeer 类可用的 doSelect 方法示例

```
// 获得 Article 对象
doSelect()

// 获得 Article 对象和 hydrate 相关作者对象
doSelectJoinAuthor()

// 获得 Article 对象和 hydrate 相关目录对象
doSelectJoinCategory()

// 获得 Article 对象和 hydrate 相关作者和目录对象
doSelectJoinAuthorAndCategory()
```

```
// 等价于
doSelectJoinAll()
```

Peer 类也包含了 doCount()的 Join 方法。有 i18n 关联的类（见第 13 章），提供了 doSelectWithI18n()方法，这个方法和 Join 方法很像不过它作用于 i18n 对象。要在模型类中发现可用的 Join 方法，你应该检查 lib/model/om/中生成的 peer 类。如果你没找到查询所需要的 Join 方法的话（例如，没有自动生成多对多关系的 Join 方法），可以自行建立并扩展你的模型。

TIP 当然，调用 doSelectJoinXXX() 会比调用 doSelect()慢些，所以只有之后需要使用化合后（hydrated）的外键对象的时候才会提高整体性能。

避免使用临时数组

当时用 Propel 时，对象已经被化合（hydrated），所以不需要在模板中准备临时数组了。开发者不习惯使用 ORM 通常导致：尽管模板可以直接依靠现有对象的数组来实现，他们还是想要准备一个字符串或者数字数组。例如，想象一下一个模板来显示从数据库中获得的所有文章主题列表的情况。一个不使用 OOP 的开发者通常会写成如例 18-6 这样。

例 18-6 - 已经有数组了，动作中再准备一个数组是没有用处的

```
// 在动作中
$articles = ArticlePeer::doSelect(new Criteria());
$titles = array();
foreach ($articles as $article)
{
  $titles[] = $article->getTitle();
}
$this->titles = $titles;

// 在模板中
<ul>
<?php foreach ($titles as $title): ?>
  <li><?php echo $title ?></li>
<?php endforeach; ?>
</ul>
```

这段代码的问题是 hydrating 已经在调用 doSelect()时完成了（需要花些时间），建立$titles 数组纯属多余，因为你能改写为例 18-7 所示的代码。因此用来建立$titles 数组的时间可以节省下来用来提高应用程序的效率。

例 18-7 - 使用对象数组可以让你不用建立临时数组

```
// 在动作中
$this->articles = ArticlePeer::doSelect(new Criteria());

// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
  <li><?php echo $article->getTitle() ?></li>
<?php endforeach; ?>
</ul>
```

如果因为一些对象的处理过程中确实需要用临时数组，正确的方法是在你的模型类中建立一个新的方法直接返回这些数组。例如，如果需要一个文章主题数组和每个文章的评论数量的话，动作和模板应如例 18-8 这样。

例 18-8 - 使用自定义的方法替代临时数组

```
// 在动作中
$this->articles = ArticlePeer::getArticleTitlesWithNbComments();

// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
  <li><?php echo $article[0] ?> (<?php echo $article[1] ?>
comments)</li>
<?php endforeach; ?>
</ul>
```

是否在模型中建立一个快速的处理过程 getArticleTitlesWithNbComments()方法取决于你——例如，通过绕过整个对象关系映射和数据库抽象层来完成。

绕过 ORM

当你确实不需要对象而只需要从一些表中获得一些字段的时候，如同以前的示例中，你能在模型中建立特殊的方法，直接通过 Creole 调用数据库完全绕过 ORM 层。例如，返回一个自定义的数组。例 18-9 说明了这个构想。

例 18-9 - 在 lib/model/ArticlePeer.php 中直接 Creole 访问来优化模型方法

```
class ArticlePeer extends BaseArticlePeer
{
  public static function getArticleTitlesWithNbComments()
  {
```

```
    $connection = Propel::getConnection();
    $query = 'SELECT %s as title, COUNT(%s) AS nb FROM %s LEFT JOIN
%s ON %s = %sGROUP BY %s';
    $query = sprintf($query,
      ArticlePeer::TITLE, CommentPeer::ID,
      ArticlePeer::TABLE_NAME, CommentPeer::TABLE_NAME,
      ArticlePeer::ID, CommentPeer::ARTICLE_ID,
      ArticlePeer::ID
    );
    $statement = $connection->prepareStatement($query);
    $resultset = $statement->executeQuery();
    $results = array();
    while ($resultset->next())
    {
      $results[] = array($resultset->getString('title'), $resultset-
>getInt('nb'));
    }

    return $results;
  }
}
```

当你开始建立这些方法的时候，你最后可能会为每个动作写一个自定义方法，这会失去层分离带来的好处，而且还失去了数据库独立性。

TIP 如果 Propel 作为模型层不适合你，在手写查询语句前考虑一下使用其他的 ORM。例如，如果想用 PhpDoctrine ORM 的话，可以看一下 sfDoctrine 插件。还有，你能用其他的数据库抽象层来代替 Creole，从而直接访问数据库。在 PHP 5.1 里，PDO 绑定在 PHP 中，而且比 Creole 快。

数据库加速

有许多针对数据库的优化技巧可以在使用 symfony 的时候用到。本节简单地列出最常用的数据库优化策略，但是良好的理解数据库引擎和管理数据库对于使用模型层会有很好的帮助。

TIP 记住网页调试工具条显示了每个页面执行查询语句的数量，应该监测每一个微调来确认是否增强了性能。

全表查询通常会发生在没有主键的列。要加速这些查询语句，你应该在数据库设计（schema）中定义索引。要增加一列索引，给列定义增加 index: true 属性，如例 18-10 所示。

例 18-10 - 在 config/schema.yml 增加一个单列索引

```
propel:
  article:
    id:
    author_id:
    title: { type: varchar(100), index: true }
```

你也可以使用另一个选择：用 index: unique 语法定义一个唯一索引替代标准的索引。你也可以在 schema.yml 中定义多列索引（关于索引语法可以参考第 8 章）。强烈建议考虑这些方法，因为这通常会对一个复杂的查询有很大的帮助。

在 schema 中增加索引后，你还需要对数据库作同样的操作，可以在数据库中直接使用 ADD INDEX 语句或是调用 propel-build-all 命令行（这不只是重建表结构，也会清空所有已存在的数据）。

TIP 索引会加速 SELECT 语句的查询效率，但会让 INSERT，UPDATE 和 DELETE 语句变慢。数据库引擎在每个查询语句只使用一个索引并且基于内部启发式的方法来推断使用哪个索引。增加索引有时会对效率带来不利的影响，所以确认你在监测效率是否有所提高。

除另有规定外，在 symfony 中每一个请求使用一个数据库连接，每一个连接在请求完成后会被关闭。在 databases.yml 文件中设置 presistent: true，可以开启持久数据库连接，这样在不同的查询之间数据库连接池会一直保持开启，如例 18-11 所示。

例 18-11 - 在 config/databases.yml 中激活永久数据库连接支持

```
prod:
  propel:
    class:          sfPropelDatabase
    param:
      persistent:   true
      dsn:          mysql://login:passwd@localhost/blog
```

这可能会增强数据库总体性能也可能不会，取决于很多因素。在因特网上关于这个主题的文档很多。请确定在修改选项前你测试过应用程序的性能来验证它的结果。

SIDEBAR 针对 MySQL 的技巧

MySQL 配置文件中的许多可以改变数据库性能的设置都放在 my.cnf 文件中。确认读过关于此主题的在线文档 (http://dev.mysql.com/doc/refman/5.0/en/option-files.html)。

MySql 提供了一个工具，慢查询记录（slow queries log）。所有 SQL 执行时间超过 long_query_time 设置（此设置可以在 my.cnf 中更改）的都会被记录在一

个文件中，这很难手动统计，但是用一个 mysqldumpslow 命令可以方便地列出总结。这是一个很棒的用来查找需要优化的查询语句的工具。

调整视图

按照不同的方法设计和实现视图层，可能会有一些小的速度减少或者提升。这小节讲述的是替代品和它们的优缺点。

使用最快的代码片段

如果没有使用缓存系统，你要注意 include_component()比 include_partial()要慢一些，include_partial()比 PHP 的 include 也要慢一些。这是因为 symfony 初始化了一个视图来包含一个局部模板和一个 sfComponent 类的对象来包含一个组件，包含这些文件会对总体性能带来一些小的影响。

不过，除非在模板中引用了许多局部模板或者组件，否则这对系统开销不是很大。这也许会发生在每次在 foreach 中调用 include_partial()辅助函数来做列表或者表格的时候。当你注意到有大量的局部模板或者组件包含非常影响性能时，应该考虑使用缓存（见第 12 章），如果不想用缓存，那只能用简单的 include 替代了。

槽（slot）和组件槽（component slot）之间的性能的差别是可以感觉得到的。设置并包含一个槽（slot）的处理时间是可以忽略的——这等于初始化一个变量。但是组件槽 （component slot）依靠一个视图配置，他们需要初始化一些对象才能工作。不过，组件槽（component slot）可以在调用模板时被单独缓存，与之相反槽（slot）总是在包含它的模板里被缓存的。

加速路由过程

正如第 9 章解释过的，在模板中每一次调用链接辅助函数都会请求路由系统来把内部 URL 转换为外部 URL。这是通过在 routing.yml 文件中查找匹配 URI 和模式来完成的。symfony 做起来很简单：它尝试用给予的 URL 去匹配第一个规则，如果不匹配，就接着尝试下一个，然后继续此步骤。由于每次测试都涉及正则表达式，这会相当耗费时间。

有一个简单的方法：使用规则名称代替模块/动作。这会告诉 symfony 使用哪一个规则并且路由系统不会花时间去尝试匹配所有前面的规则。

在具体的条件中，假设定义在 routing.yml 文件中的路由规则如下：

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read }
```

然后用这个方法把输出的超连接替换掉：

```
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>
```

你应该用最快实现的方法：

```
<?php echo link_to('my article', '@article_by_id?id='.$article->getId()) ?>
```

注意只有在包含了很多路由链接的页面里，这种差别才会比较明显。

略过模板

通常，一个回应是由一组头信息和内容组成的。但是有些回应不需要内容。例如，一些Ajax交互只需要从数据库获得一些数据并提供给 JavaScript 程序用来更新页面的不同部分。对于这些短回应，一套单独的头信息会更适合传递。如第11章讨论的，一个动作只能返回一个JSON 头。例18-12重现了第11章的一个例子。

例 18-12 - 动作返回一个JSON头信息的示例

```
public function executeRefresh()
{
  $output = '<"title", "My basic letter"], ["name", "Mr Brown">';
  $this->getResponse()->setHttpHeader("X-JSON", '('.$output.')');

  return sfView::HEADER_ONLY;
}
```

这跳过了模板和布局，可以立即发出回应。由于它仅包含头，这会更轻巧，并会用较少时间传递给用户。

第6章解释了另一个跳过模板的方法，就是直接从动作返回内容文字。这就打破了MVC的规则，但这能显著提高动作的响应速度。看例18-13的示例。

例 18-13 - 动作直接返回内容的示例

```
public function executeFastAction()
{
  return $this->renderText("<html><body>Hello, World!
</body></html>");
}
```

限制默认的辅助函数

标准的辅助函数组(局部模板 Partial, 缓存 Cache 和表单 Form)在每次请求的时候都会载入。如果你确认你不使用它们中的一些，从标准列表中移除一个辅助函数组会节省解析辅助函数文件的时间。特别是表单 Form 辅助函数组，尽管默认包含了，但是因为他的大小，还是会减慢没有表单的页面的时间。所以在 settings.yml 文件的 standard_helpers 设置中去掉它也许是个好办法：

```
all:
  .settings:
    standard_helpers: [Partial, Cache]     # Form 被移除
```

相对的，但是你必须在每一个使用 Form 辅助函数组的模板中使用 use_helper('Form')。

压缩回应

symfony 在发送给用户回应前压缩了相应内容。这个功能基于 PHP 的 zlib 模块。你可以在 settings.yml 文件中关闭这个选项来获得一些 CPU 时间：

```
all:
  .settings:
    compressed: off
```

要注意获得 CPU 时间会损失带宽，所以并不是在所有的配置中改变这个设置都会增加性能。

TIP 如果关闭了 PHP 的 zip 压缩功能，你可以在服务器层激活它。Apache 有他自己的压缩扩展。

调整缓存

第 12 章已经说过如何缓存部分或者全部回应。回应缓存会带来很大的性能提升，这应该是最优先考虑的优化。如果你想要最大化地利用缓存系统，进一步阅读本章节来了解一些你未曾想过的技巧。

选择性的清除部分缓存

在应用程序开发过程中，你必须在一些环境中清除缓存：

- 当建立一个新类： 在自动载入目录中增加一个类（在项目的 lib/目录下）是无法让 symfony 自动找到它的。你必须清除自动载入配置缓存才能让 symfony 再次浏览所有 autoload.yml 文件中定义的目录并引用可自动载入类的位置——这才能包含新建的类。
- 当你在生产环境中修改了配置文件的时候： 配置文件只在生产环境的第

一次请求的时候会被解析。其余的请求使用的是缓存了的配置文件。 所以在生产环境中修改的配置文件（或者任何 SF_DEBUG 是 off 状态下的环境）只会在清除缓存后才会生效。

- 当你在模板缓存已经激活的环境中修改了模板的时候： 在生产环境中，有效的缓存模板总是会替代已经存在的模板而优先得到，所以只有模板缓存被清空后模板的修改才会生效。
- 当用 sync 命令行去更新应用程序的时候： 这通常包括了前面三个修改。

清除所有缓存会带来一个问题，因为需要生成配置缓存，所以下一个请求会需要花较长的时间来处理。除此之外，未修改过的模板缓存也会被清除掉，这样就会失去之前的请求中缓存带来的速度提升。

这就是说最好的方法是只清除需要重新生成的那部分缓存。使用 clear-cache 去定义哪些缓存需要清除，如例 18-14 这样。

例 18-14 - 有选择的清除部分缓存

```
// 只清除 myapp 应用程序缓存
> symfony clear-cache myapp

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

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

你也可以手动删除 cache/目录下的文件，或者有选择的通过$cacheManager->remove()方法来清除模板缓存，如第 12 章所述。

上面列出的所有这些技巧会使性能的负面影响最小化。

TIP 当升级了 symfony，缓存会自动清除（如果在 settings.yml 中设置了 check_symfony_version 参数为 true）。

生成缓存页

当你部署一个新的应用程序到生产服务器上的时候，模板缓存是空的。你必须等待用户访问页面一次让页面生成缓存。在一些关键的部署中，生成页面的系统开销是无法接受的，在第一次请求之前必须生成缓存。

解决办法是在临时工作环境（staging)中（配置文件和生产服务器上很相似）自动浏览应用程序的页面从而生成模板缓存，然后把应用程序和缓存一起放到生产服务器上。

要去自动浏览页面，一个办法是建立一个 shell 脚本调用浏览器（例如 curl）

依次访问外部连接。但是有一个更好更快的解决方案：使用 sfBrowser 对象的一个 symfony 批处理，这在第 15 章已经讨论过。这是一个 PHP 写的内部浏览器（使用 sfTestBrowser 来做功能测试）。它访问外部 URL 并返回一个结果，但有趣的是这会像用正常浏览器访问一样生成模板缓存。因为他只是初始化一次 symfony 而且并不通过 HTTP 传送层传递，这个方法会更快一些。

例 18-15 展示了一个批处理脚本的示例，用来在临时工作环境（staging)中生成模板缓存文件。用 php batch/generate_cache.php 开始这个缓存过程。

例 18-15 - 在 batch/generate_cache.php 生成模板缓存

```
<?php

define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/..'));
define('SF_APP',         'myapp');
define('SF_ENVIRONMENT', 'staging');
define('SF_DEBUG',       false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');

// 需要去浏览的 URL 数组
$uris = array(
  '/foo/index',
  '/foo/bar/id/1',
  '/foo/bar/id/2',
  ...
);

$b = new sfBrowser();
foreach ($uris as $uri)
{
  $b->get($uri);
}
```

使用数据库存储作为缓存

symfony 默认使用文件系统作为模板缓存的：HTML 块或者回应对象序列化之后储存在项目的 cache/目录下。symfony 建议另一个方法来储存缓存：一个 SQLite 数据库。这是一个 PHP 原生的，可以非常有效实现查询的简单文件数据库。

要让 symfony 使用 SQLite 储存代替文件系统储存模板缓存的话，打开

factories.yml 文件并编辑 view_cache：

```
view_cache:
  class: sfSQLiteCache
  param:
    database: %SF_TEMPLATE_CACHE_DIR%/cache.db
```

使用 SQLite 储存作为模板缓存的好处是当缓存元素数量很关键的时候可以更快速的做读写操作。如果你的应用程序的缓存压力非常大，模板缓存文件最终会分散在很深的文件结构中； 在这时候，用 SQLite 存储会更快。另外，在文件系统存储中清除缓存会有一个从磁盘删除很多文件的动作； 这个操作会持续好几秒，这时应用程序是无法访问的。使用 SQLite 存储系统的话，清除缓存过程将只是一个简单的文件删除操作：删除 SQLite 数据库文件。无论缓存元素数据有多大，操作瞬间就会完成。

绕过 symfony

也许最好的加速 symfony 的方法是完全绕过它……这不是一个玩笑。在每次请求中会有一些页面由于没有更改过所以不需要重新由框架来处理。模板缓存已经在加速传递页面了，但是这依旧是依靠 symfony 来处理的。

在第 12 章说过的一些小技巧允许一些页面完全绕过 symfony。第一个是对页面缓存本身请求代理服务器和客户端浏览器做缓存，包含了使用 HTTP 1.1 头文件，所以当这个页面需要的时候他们不需要再次请求了。第二个就是极速缓存（由 sfSuperCachePlugin 插件自动完成），这包含了在 web/目录中储存一份回应的副本和修改重写规则，这样 Apache 会在把请求指向 symfony 前先会查看缓存版本。

尽管他们只是针对静态页面但上述两种方法都非常有效，它们将为 symfony 分担这些页面的处理，这也会让服务器能全力处理其他请求。

缓存函数调用的结果

如果一个函数不是环境敏感的值也不是随机调用的话，用相同的参数调用它两次应该返回同一个结果。这就是说当第二次调用的时候如果第一次结果已经储存下来的话就可以避免再次调用它。这就是 sfFunctionCache 类做的事情。这个类有一个 call()方法，可以通过输入一些参数来调用。当被调用的时候，这个方法用所有他的参数建立一个 md5 哈希值作为名字并在缓存目录下找此名字的文件。如果文件找到了，此方法就会返回存在文件中的结果。如果没有，sfFunctionCache 就执行这个函数，并把结果储存在缓存中，并返回值。所以第二次执行例 18-16 会比第一次执行更快。

例 18-16 - 缓存函数结果

```
$function_cache_dir = sfConfig::get('sf_cache_dir').'/function';
$fc = new sfFunctionCache($function_cache_dir);
$result1 = $fc->call('cos', M_PI);
$result2 = $fc->call('preg_replace', '/\s\s+/', ' ', $input);
```

sfFunctionCache 的构造函数需要一个绝对路径作为参数（该目录必须在对象初始化之前就存在）。call()方法的第一个参数必须是 PHP 调用名，所以它可以是一个函数名，一个类名字的数组，静态方法名字，对象名字的数组或者公共方法名。你能用任意多的其他参数作为 call()的参数——它们都会被作为调用的参数。

这个对象对很消耗 CPU 的函数特别有用，因为文件 I/O 的开销超过处理一个简单函数的时间。它依赖于 sfFileCache 类，这也是模板缓存引擎的一个组件。详情请查阅 API 文档。

CAUTION clear-cache 任务只是删除 cache/下文件。如果函数缓存储存在其他地方，通过命令行执行这个命令的时候不会被自动清除。

在服务器上缓存数据

PHP 加速器提供了一些特别的函数在内存中储存数据，因此你可以再次通过它处理请求。问题是他们都用了一些不同的语法，每一个都用自己的方法来处理这个任务。symfony 提供了一个叫做 sfProcessCache 的类用来抽象化所有的这些不同的工作而不管你用的是什么加速器。参见例 18-17 的语法。

例 18-17 - sfProcessCache 方法的语法

```
// 在 Process 缓存中存储数据
sfProcessCache::set($name, $value, $lifetime);

// 获得数据
$value = sfProcessCache::get($name);

// 检查 process 缓存中是否有此数据
$value_exists = sfProcessCache::has($name);

// 清除 process 缓存
sfProcessCache::clear();
```

如果缓存不工作的话 set()方法会返回 false。 缓存的值可以是任意的（一个字符串，一个数组，一个对象）； sfProcessCache 类会处理序列化的过程。如果缓存中没有需求的值，get 方法会返回 null。

甚至在没有安装加速器的情况下 sfProcessCache 类的方法依旧会工作。因此，

只要你提供一个返回值，尝试从 process 缓存中获得数据总是安全的。例如，例 18-18 显示了如何从 process 缓存中获得参数设置的过程。

例 18-18 - 安全的使用 Process 缓存

```
if (sfProcessCache::has('myapp_parameters'))
{
  $params = sfProcessCache::get('myapp_parameters');
}
else
{
  $params = retrieve_parameters();
}
```

TIP 如果你想更进一步了解内存缓存，仔细的阅读一下 PHP 的 memcache 扩展。它能帮助在负载均衡的应用程序中减少数据库负载，并且 PHP5 提供了它的 OO 接口(http://www.php.net/memcache/)。

屏蔽未使用过的功能

默认的 symfony 配置激活了大多数网页应用程序常用的功能。然而，如果你不想要所有的这些，你可以屏蔽他们从而在每个请求中节省初始化的时间。

例如，如果你的应用程序不使用用户会话机制，或者你想手动处理用户会话，你应该将 factories.yml 文件的 storage 键值 auto_start 设置为 false，如例 18-19 所示。

例 18-19 - 在 myapp/config/factories.yml 中把用户会话关闭

```
all:
  storage:
    class: sfSessionStorage
    param:
      auto_start: false
```

同样的对于数据库（如先前讨论的“调整模型”）和转义输出功能（见第 7 章）。如果应用程序不需要使用他们，屏蔽他们会让系统效率有些许提升，他们的设置在 settings.yml 文件中（见例 18-20）。

例 18-20 - 在 myapp/config/settings.yml 中屏蔽功能

```
all:
  .settings:
    use_database:       off     # 数据库和模型功能
```

```
    escaping_strategy: off     # 输出转义功能
```

关于安全和短暂属性功能（见第 6 章），你可以在 filters.yml 文件中屏蔽他们，如例 18-21 所示。

例 18-21 - 在 myapp/config/filters.yml 中屏蔽功能

```
rendering: ~
web_debug: ~
security:
  enabled: off

# generally, you will want to insert your own filters here

cache:     ~
common:    ~
flash:
  enabled: off

execution: ~
```

一些功能只是在开发过程中有用处，所有开发过程中最影响性能的就是 SF_DEBUG 模式了。所以你应该在生产环境中屏蔽他们。默认就是这么做的，因为 symfony 的生产环境已经优化过性能了。还有就是 symfony 日志，这个功能已经在生产环境中默认关闭了。

你也许会想在日志关闭的时候如何在生产环境中得到错误信息，并认为这个问题并不只出现在开发过程中。幸运的是，symfony 可以使用 sfErrorLoggerPlugin 插件，用来在生产环境后台中记录 404 和 500 错误到数据库中。这比写入文件日志功能更快，因为插件方法只在请求失败时候被调用，当日志机制打开后，不论在什么层次都增加了一个不可忽视的开销。这个插件的安装指南和操作手册网址是 http://www.symfony-project.com/trac/wiki/sfErrorLoggerPlugin。

TIP 要确保经常检查服务器错误记录——他们也许有关于 404 和 500 错误的非常有用的信息。

优化你的代码

优化代码本身也可以加速你的应用程序。本节提供一些改进的好意见。

编译核心

载入 10 个文件需要比载入一个大文件花费更多的 I/O 操作，特别是在低速磁盘中。载入一个非常大的文件需要比载入一些小文件占用更多的资源——特别是

文件的很大一部分不需要使用 PHP 解析器的时候，例如注释。

因此合并大量的文件并且把它们的注释删除是一个很好的增强性能的方法。symfony 已经做了优化；这就是所谓的核心编译。在第一个请求开始的时候（或者在缓存清空后），一个 symfony 应用程序合并所有的核心框架类（sfActions, sfRequest, sfView 和其他）到一个文件中，删除了注释和多余的空格来优化文件大小，并把它存入缓存中，取名为 config_core_compile.yml.php。每一个接下去的请求只是读取了这个优化过的文件。

如果你的应用程序有类需要加载，尤其是有一个庞大的包含了很多注释的类，把他们加入核心编译文件会很有好处。要这么做的话，只要在应用程序的 config/目录下增加一个 core_compile.yml 文件，列出所有需要增加的类，就如例 18-22 一样。

例 18-22 - 在核心编译文件 myapp/config/core_compile.yml 中增加你的类

```
- %SF_ROOT_DIR%/lib/myClass.class.php
- %SF_ROOT_DIR%/apps/myapp/lib/myToolkit.class.php
- %SF_ROOT_DIR%/plugins/myPlugin/lib/myPluginCore.class.php
...
```

sfOptimizer 插件

symfony 也提供了其他的优化工具，叫做 sfOptimizer。它把许多优化策略应用到了 symfony 和应用程序代码中，用来加速执行效率。

symfony 代码依赖于很多依靠配置文件参数的测试——你的应用程序也是这么做的。例如，如果你看一下 symfony 类，你会经常看到在调用 sfLogger 对象前会有一个带有 sf_logging_enabled 的测试值：

```
if (sfConfig::get('sf_logging_enabled'))
{
   $this->getContext()->getLogger()->info('Been there');
}
```

尽管 sfConfig 注册表已经很好的优化过了，但在每一次处理请求调用它的 get()方法的次数还是很重要的——这会影响最终的性能。sfOptimizer 的一个优化技巧是用配置常量的值替换它们本身，只要这些常量在运行时不变。例如，用 sf_logging_enable 参数； 当它定义为 false 的时候，sfOptimizer 会把它转换为：

```
if (0)
```

```
{
   $this->getContext()->getLogger()->info('Been there');
}
```

另外，之前的这个例子里，配置值如果是空字符串也会有这样的优化结果。

要用到这个优化，你必须先安装插件 http://www.symfony-project.com/trac/wiki/sfOptimizerPlugin 然后调用 optimize 任务，制定一个应用程序和环境：

```
> symfony optimize myapp prod
```

如果你想用到其他的优化策略，sfOptimizer 插件应该是一个好的开始。

总结

symfony 是一个已经优化得非常好的框架了，能用来处理高访问量网站。但如果你确实需要优化你的应用程序性能，调整配置文件（无论是服务器配置，PHP 配置或者是应用程序设置）会带来一些小的加速。你也应该遵循好的策略来写有效的模型方法；因为数据库通常是网页应用程序的瓶颈，这点应该引起你的注意。模板总能用一些小技巧来优化，但最好的加速方法是用缓存。最后，不要犹豫，去看看已经有的插件，这些插件会提供一些创新的技巧来加速你的网页的（如：sfSuperCache, sfOptimizer）。