Django 性能优化

Django 官网（En）：https://docs.djangoproject.com/en/1.10/topics/performance/； 下文为编译 by @flora5 [github]

Performance and optimization

本文档提供了有助于使 Django 代码更高效，更快速运行，并使用更少系统资源的技术和工具的概述。

Introduction

通常，首先关心的是编写能工作的代码，逻辑函数输出符合预期。 但这不能保证效率。

需要一系列提高代码性能的方法，而不会或尽少的影响代码行为。

General approaches

What are you optimizing for?

速度，CPU，内存，数据存储，还是网络占用。

一项性能的改进通常会使其他方面的性能也被提升（比如：传输的数据变小，存储量变小，网络带宽压力也变小），但有时候，一项性能的提升是以另一项的下降为代价的。

比如：提升程序的速度可能导致占用内存更多，甚至弄巧成拙，提升速度占用的内存过大，导致系统内存用尽 。

你的时间比 CPU 时间宝贵，一些改进可能太难，不值得执行，或者影响代码的可移植性，可维护性。不是所有的改进都值得去做。要明确改进哪方面的性能，改进的理由。

Performance benchmarking

不能只通过猜测和假设来确定代码哪里性能差。

Django tools

django-debug-toolbar 是一个很方便的工具，能检查代码做了什么，做这些事情所用的时间，特别是能报告页面上产生的所有 SQL 查询， 每个查询用了多少时间，第三方面板（Third-party panels ）也可用于 toolbar，可以报告 cache 的性能和模板 render 的时间。

Third-party services

一些免费服务可以从远程 HTTP 客户端角度分析和报告网站性能，模拟真实用户体验，虽然不能报告代码内部的状况，但可以洞察网站的整体性能，包括不能从 Django 环境中充分衡量的方面。例如：

Yahoo 的 Yslow

Google PageSpeed

还有一些付费服务提供类似的分析，包括一些 Django 能识别的（Django-aware），能跟代码库集成，以便更全面分析其性能。

Get things right from the start

一些优化工作涉及解决性能缺陷，但是有些工作可以简单地建立在你所做的工作中，作为你应该采取的良好做法的一部分，甚至在你开始考虑提高性能之前就做这些工作。

在这方面 python 是很好的语言，因为看起来优雅，感觉正确，的解决方案通常也是性能最好的那个，跟其他的技能一样，学习什么是「看起来正确」的需要练习，但最有用的指南之一是：

Work at the appropriate level

Django 提供了不同的方法来处理事情，但能用的方式不一定是最合适的。例如， 统计一个 QuerySet 的 collection 里 items 的数量， 可以在 python 里或者在 template 里，在较低级别几乎总是比在较高级别快，

在更高层次，系统必须同多层抽象和多层机械（machinery）来处理对象。数据库处理东西通常能比在 python 中快， python 中比在模板语言中快.

```
# QuerySet operation on the database
# 更快，因为这是数据库擅长做的
my_bicycles.count()
# counting Python objects
# 更慢, 因为需要调用数据库查询，加上 python 对象的解析
len(my_bicycles)
# Django template filter
# 更慢,因为要在 python 中计算，还有模板语言的开销
{{ my_bicycles|length }}
```

一般来说，最合适的层是，适合编写代码的层当中，最低的那层

Note:

> 实际还有要考虑在 count 之前和之后发生了什么，在特定上下文中什么是最佳的方法。 Django 的数据库优化文档，描述了一个在模板中哪里执行 counting 更好的例子。
>
> 其次还有其他选择要考虑:
>
> 在实际情况下，{{ my_bicycles.count }} 直接从 template 调用 QuerySet count() 方法，可能是最合适的选择。

Cacheing

值计算是很占用资源的，速度慢， 将值存在可以快速访问的缓存中，方便下次调用能明显提升性能

Django 包含了一个完整的 caching framework，和一些小的缓存功能

The caching framework

Django 的缓存框架通过保持动态内容以便不需要为每个请求重新计算，为性能提升提供了重要的机会。

为方便, Django 提供了不同级别的缓存粒度：可以缓存特定 view 的输出, 或者只缓存难以生成的片段，或者缓存整个网站；

实现缓存不应该看做改进因为写的差而性能差的代码的可选方案。 它是提供性能良好的代码的最后一步，而不是捷径。

cached_property

通常必须多次调用类实例方法，如果这个方法很占资源，这将很浪费。

用 cached_property 装饰器保存属性返回的值； 下次函数在实例上调用函数时 ，将返回保持的值，而不是重新计算它。注意： 这只适用于以 self 作为唯一参数的方法。and that it changes the method to a property.

特定的 Django 组件也有它自己的缓存功能； 这些将在下面跟模块相关的部分介绍

Understanding laziness

Laziness 对 caching 来说是策略互补。 Caching 通过存储结果避免重新计算, laziness 将计算推迟到真正必要的时候. Laziness 允许我们在实例化之前-甚至可能实例化之前引用它，这有很多用途，例如 ：

lazy translation 可以用在目标语言确定之前，因为在 translated string 实际被使用之前，延迟翻译它不会占用空间，像在渲染了的模板中一样。 *（详见 Translation 文档）*。

Laziness 也是一种在初期避免工作的省力方式，laziness 意思之一是：必须时才做，因为最后可能根本用不到。

因此对性能有影响，任务的开销越大，通过 Laziness 获得的性能提升也越大。

Python 提供了一些 lazy evaluation 的工具，尤其通过 generator 和 generator expression 结构，阅读 python 的 laziness， 去发现在你的代码中使用 lazy patterns 的机会是值得的。

Laziness in Django

Django is itself quite lazy，可以在 QuerySets 的求值中找到一个很好的例子；

QuerySets are lazy.

因此可以创造，传递，和与其他 QuerySets 组合的 QuerySet，而不实际引发任何到数据库的访问，以获取它描述的 items。 来回传递的是 QuerySet 对象，不是最终需要从数据库中获取的项的集合（collection of items） 。

另一方面, 某些操作将强制 QuerySet 求值。避免对 QuerySet 过早求值可以节省对数据库昂贵和不必要的访问；

Django 还提供了一个 keep_lazy() 装饰器. 这允许 用延迟参数调用的函数，本身变的 lazy， 只在必须的时候才进行求值。

Databases

Database optimization

Django 的数据库层提供了各种帮助开发者在数据库获得最佳性能的方法

数据库优化文档汇集了相关文档的链接，并添加了各种提示，列出了尝试去优化数据库使用时需要采取的步骤

Other database-related tips

对相当大一部分请求处理时间，启用持久连接可以加速到数据库账户的连接，这对网络性能有限的 virtualized hosts 很有帮助

HTTP performance

Middleware

Django 提供了一些有用 middleware 帮助优化网站性能，包括:

ConditionalGetMiddleware

添加了对现代浏览器的支持，以根据 ETag 和 Last-Modified 标头有条件地“GET” 响应。

> 补充： conditional GET 请求可能返回 304 而不是 200， 304 响应表示资源从上次被访问后，没有修改过，资源不用返回给客户端，客户端用缓存的内容就可以；减少了带宽浪费

有至少两种（不完全独立的）方法实现 conditional GET:

1，Last-Modified / If-Modified-Since

2，ETag / If-None-Match

GZipMiddleware

为所有现代浏览器压缩 responses， 节约带宽和传输时间。 GZipMiddleware 目前被认为有安全风险，容易受到由 TLS / SSL 提供的保护无效的攻击。 有关详细信息，请参阅 GZipMiddleware 中的警告。

Sessions

Using cached sessions

用缓存的 sessions 可能是一个提高性能的方式。 将常用的 session 数据存在内存中， 从而省去从低速存储源，如数据库，加载 session 数据的需求

Static files

静态文件对优化来说是很好的目标。

CachedStaticFilesStorage

利用浏览器缓存，在初始下载之后，可以完全的消除给定文件的网络点击（eliminate network hits）

CachedStaticFilesStorage 会在静态文件的文件名中附加一个内容相关的标签，以使浏览器能够安全地对其进行长期缓存，而不会错过将来的更改 – 当文件更改时，标签也将更改，因此浏览器将自动重新加载资源。

“Minification”

一些第三方 Django 工具和包提供了“minify”（压缩） HTML, CSS, 和 JavaScript 的功能，去掉不必要的空格，换行符，注释，缩短变量名称，来减少你的网站发布的文档的大小。

Template performance

用 {% block %} 比用 {% include %}快， 高度碎片化的 templates,由许多小块组成，会影响性能。

The cached template loader

启用 cached template loader 经常会显著改善性能，因为它避免在每次需要渲染的时候重新编译

Using different versions of available software

有时候需要检查, 是不是有性能更好的软件版本可用。

这些技术是针对更高级的用户的，这些用户希望提高已经优化的很好的 Django 网站的性能, 这不是魔法方法，对一个没有将更基本的事情做对的网站， 不会带来更好的边际收益 。

Note

> 值得重复的是：寻求软件的替代品，永远不是解决性能问题的第一个答案。 当达到这种优化级别的时候，需要一个正式的 benchmarking （数据对比）

Newer is often - but not always - better

维护良好的软件很少出现新版本性能比旧版本低的情况，但是开发人员无法预料所有可能的使用场景 - 因此，虽然知道较新的版本可能性能更好，但不要简单地假设一直会如此。

Django 已经在连续版本中做了许多改进， 但仍应检查应用程序在真实场景中的性能，因为在某些情况下，可能会发现这些更改意味着性能更差，而不是更好。

较新的 Python 版本以及 Python 软件包，通常会表现得更好 - 但是：应该测量而不是假设。

Note

> 除非您在特定版本中遇到了异常的性能问题，否则在新版本中通常会找到更好的功能，可靠性和安全性，这些优点远远胜过您可能获胜或失败的任何性能。

Alternatives to Django's template language

大部分时候，Django 内置的模板语言足够用。 但是如果 Django 项目的瓶颈似乎在于模板系统，已经用尽了其他方式来弥补这一点，可以考虑用第三方模板引擎。

Jinja2 能提供了性能改进，尤其在速度上，备选模板系统在共享 Django 的模板语言程度上各不相同。

Note

> 如果在模板中遇到性能问题，首先要做的是了解为什么
>
> 使用其他可选模板系统可能会更快，但是也可以不用那么麻烦就获得相同的性能提升 – 例如：更好的处理，Template 中的逻辑会在 view 中执行的更快。

Alternative software implementations

检查所使用的 Python 软件是否提供了不同的实现，能更快的执行相同的代码。

编写良好的 Django 网站，性能问题大多不是在 Python 执行层面，而是在于低效的数据库查询，缓存和模板。如果依赖写得不好的 Python 代码，性能问题不可能通过使他更快地执行得到解决。

使用其他替代实现可能引入兼容性，部署，可移植性或维护问题。 在采用非标准实现之前，应该确保它为应用程序提供了足够的性能提高，超过潜在的风险。

PyPy

PyPy 是用 python 实现的 python（python 标准实现是 C），通常对大型应用 PyPy 能提供显著的性能提升。PyPy 项目的一个关键目标是兼容现有的 Python API 和库。 Django 是兼容的，但需要检查依赖的其他库是否兼容。

C implementations of Python libraries

一些 python 库也是用 C 语言实现的， 速度更快； 目标是提供相同的 API。