**$ ruby class\_check1.rb**

0.022767841

chp2/class\_check2.rb

require *'benchmark'*

obj = *"sample string"*

time = Benchmark.realtime **do**

100000.times **do**

obj.is\_a?(String)

**end**

**end**

puts time

**$ ruby class\_check2.rb**

0.019568893

在一个中等大的循环中，再次进行100,000次迭代，这样的检查需要19-22 ms。这听起来并不糟糕，但是，例如，Rails应用程序可以为每个请求调用比较运算符超过100万次，并且花费超过200毫秒进行类型检查。

将类型检查从迭代器或经常调用的函数和运算符移开是个好主意。如果你不能，不幸的是，你无能为力。

BigDecimal::==(String)

从数据库获取数据的代码大量使用大小数。对于Rails应用程序尤其如此。此类代码通常从它从数据库中读取的字符串创建BigDecimal，然后将其直接与字符串进行比较。

问题是，在Ruby 1.9.3及更低版本中进行此比较的自然方式令人难以置信地缓慢：

chp2/bigdecimal1.rb

require *'bigdecimal'*

require *'benchmark'*

x = BigDecimal(*"10.2"*)

time = Benchmark.realtime **do**

100000.times **do**

x == *"10.2"*

**end**

**end**

puts time

**$ rbenv shell 1.9.3-p551**

**$ ruby bigdecimal1.rb**

0.773866128

**$ rbenv shell 2.0.0-p598**

**$ ruby bigdecimal1.rb**

0.025224029

**$ rbenv shell 2.1.5**

**$ ruby bigdecimal1.rb**

0.027570681

**$ rbenv shell 2.2.0**

**$ ruby bigdecimal1.rb**

0.02474011096637696

较旧的Rubys具有令人无法接受的BigDecimal :: ==函数的慢实现。这个性能问题随着Ruby 2.0升级而消失。但如果你无法升级，请使用这个聪明的技巧。在比较之前将BigDecimal转换为String：

chp2/bigdecimal2.rb

require *'bigdecimal'*

require *'benchmark'*

x = BigDecimal(*"10.2"*)

time = Benchmark.realtime **do**

100000.times **do**

x.to\_s == *"10.2"*

**end**

**end**

puts time

**$ rbenv shell 1.9.3-p545**

**$ ruby bigdecimal2.rb**

0.195041792

这个hack比Ruby 2.x实现快三到四倍 - 不快四十倍，但仍然是一个改进。

Write Less Ruby

教我编程的大师之一曾经说过最好的代码是不存在的代码。如果我们可以在不编写任何代码的情况下解决问题，那么我们就不必对其进行优化。对？

不幸的是，在现实世界中，我们仍然编写代码来解决我们的问题。但这并不意味着它必须是Ruby代码。其他工具更好地做某些事情。我们已经看到Ruby在两个方面特别糟糕：大型数据集处理和复杂计算。那么让我们看看你可以使用什么，以及它如何提高性能。

Offload Work to the Database将工作卸载到数据库

Ruby社区倾向于仅将数据库视为数据存储工具。Rails开发人员特别容易这样做，因为他们经常使用ActiveRecord和ActiveModel抽象而无需直接与数据库连接。

所以，是的，你可以在不知道任何SQL或理解MySQL和PostgreSQL之间的差异的情况下构建Rails应用程序。但通过这样做，您将为了方便而交易性能，并错过数据库提供的数据处理能力。

事实证明，数据库非常擅长复杂的计算和其他类型的数据操作。让我告诉你他们有多好。

让我们想象一下，我们有一个庞大的数据库，公司员工，例如，在25个不同部门工作的10,000人。我们知道每个人的工资，并且我们想要按工资计算员工在部门内的等级。

我将在这个例子中使用PostgreSQL，并且为了简单起见将创建随机数据。

要重现此示例，您应该安装并启动PostgreSQL数据库服务器。

**$ createdb company\_data**

**$ psql company\_data**

**create table** empsalaries(

department\_id **integer**,

employee\_id **integer**,

salary **integer**);

**insert into** empsalaries (

**select** (1 + round(random()\*25)), \*, (50000 + round(random()\*250000))

**from** generate\_series(1, 10000)

);

**create index** empsalaries\_department\_id\_idx **on** empsalaries (department\_id);

如果您不熟悉PostgreSQL，请解释一下。insert语句将生成一系列10,000行（我们的员工ID），然后为每个行分配一个从1到25的随机部门ID和一个从$ 50,000到$ 250,000的随机工资。

我们首先使用ActiveRecord来计算员工排名。为此，我们将创建一个名为group\_rank的文件夹，其中包含Gemfile和group\_rank.rb。

chp2/group\_rank/Gemfile

source 'https://rubygems.org'

gem 'activerecord'

gem 'pg'

chp2/group\_rank/group\_rank.rb

require 'rubygems'

require 'active\_record'

ActiveRecord::Base.establish\_connection(

:adapter => "postgresql",

:database => "company\_data"

)

class Empsalary < ActiveRecord::Base

attr\_accessor :rank

end

time = Benchmark.realtime do

salaries = Empsalary.all.order(:department\_id, :salary)

key, counter = nil, nil

salaries.each do |s|

if s.department\_id != key

key, counter = s.department\_id, 0

end

counter += 1

s.rank = counter

end

end

puts "Group rank with ActiveRecord: %5.3fs" % time

现在让我们运行bundler来安装所有需要的gems并启动applica-查看执行时间：

$ cd group\_rank

$ rbenv shell 2.2.0

$ bundle install --path .bundle/gems

$ bundle exec ruby group\_rank.rb

Group rank with ActiveRecord: 0.264s

仅用246毫秒处理10000行是相当糟糕的。现在试着去做10万行和100万行也是一样。Ruby>=2.0需要分别为2.4秒和24秒。像1.8和1.9这样的老鲁比甚至可能不会结束，因为GC会太频繁地启动。我有足够的耐心等110Ruby1.9处理一百万行的秒数。我很肯定我的代码不是那个病人。

现在让我们看看PostgreSQL在10000行代码上做相同的事情可以多快：

$ psql company\_data

=# \timing

Timing is on.

=# select department\_id, employee\_id, salary,

rank() over(partition by department\_id order by salary desc)

from empsalaries;

Time: 22.573 ms

这在PostgreSQL中是10倍。作为奖励，它还可以很好地扩展。它10万行需要280毫秒，100万行需要2.3秒。

请注意，PostgreSQL的性能始终比Ruby的最快的快十倍。

是的，我的例子使用了Postgres特有的特性，比如window功能。但这正是我的观点。数据库在数据处理方面要好得多。这就大不一样了。我们已经看到十次不是极限。有时候这是天壤之别，在Ruby中永远不完成任务数据库可以在几秒钟内完成它，只需让数据库做它所擅长的。

在C中重写

Ruby是在C语言中实现的，因此它有一种简单的方法来与C代码进行接口。所以如果您的Ruby代码很慢，您可以随时用C重写它。等等！什么？恐惧

不，我不会试图说服你自己写C代码。你当然也可以这么做，但这不在本书的范围之内。相反，我想指出有很多用C语言写的Ruby比他们的对手做的快。

我将这些本地代码宝石分为两种类型：

1。在C语言中重写ruby或ruby-on-rails的缓慢部分的gems

2。在C语言中实现特定任务的gems

date:：performance gem 3是第一种类型的一个很好的例子。这是一条古老的代码，所有Ruby1.8开发人员都应该使用。它用C编写的类似代码取代了慢Ruby日期和日期时间库。

请注意，date:：performance gem仅为ruby 1.8。Ruby 1.9及更高版本有一个更快的日期库。

让我展示一下date:：performance有多快。为此，我们将切换到ruby1.8，安装日期性能gem，并测量创建大量日期对象的程序的执行时间（不使用gc，将其分解）。

$ rbenv shell 1.8.7-p375

$ gem install date-performance

Fetching: date-performance-0.4.8.gem (100%)

Building native extensions. This could take a while...

Successfully installed date-performance-0.4.8

1 gem installed

让我们看看标准库中的日期是如何执行的。

chp2/date\_without\_date\_performance.rb

require 'date'

require 'benchmark'

GC.disable

memory\_before = `ps -o rss= -p #{Process.pid}`.to\_i/1024

time = Benchmark.realtime do

100000.times do

Date.new(2014,5,1)

end

end

memory\_after = `ps -o rss= -p #{Process.pid}`.to\_i/1024

puts "time: #{time}, memory: #{"%d MB" % (memory\_after - memory\_before)}"

$ ruby date\_without\_date\_performance.rb

time: 2.19644594192505, memory: 262 MB

我们需要2.2秒来创建100000个日期。现在让我们将其与Date::Performance进行比较。

chp2/date\_with\_date\_performance.rb

require 'benchmark'

require 'rubygems'

require 'date/performance'

GC.disable

memory\_before = `ps -o rss= -p #{Process.pid}`.to\_i/1024

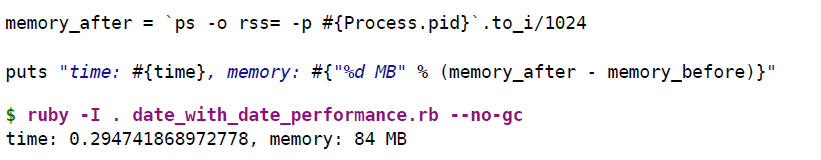
time = Benchmark.realtime do

100000.times do

Date.new(2014,5,1)

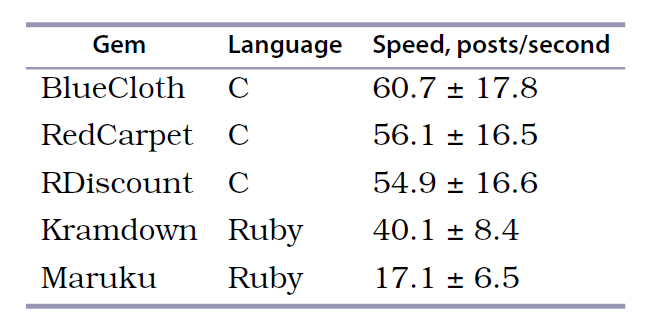
end

end



用C编写的相同代码几乎快八倍！ 作为奖励，它使用175 MB（或更少）的内存。 两者都有很大的改进。 这就是为什么我建议每个坚持使用旧版Ruby 1.8的人都应该使用Date :: Performance gem的原因。

还有一些gems在C中实现特定的任务。最好的例子是markdown libraries。 有些是用C语言编写的，有些是用Ruby编写的。 以下是Jekyll博客引擎贡献者之一Jashank Jeremy的性能比较：



最慢的C实现（RDiscount）比最快的Ruby实现（Kramdown）快1.4倍。 最快和最慢之间的差异是惊人的的3.5倍。 正如你所看到的，在本机代码中搜索gems这项困难的任务是完全有意义的。

**小贴士**

我们在本章中看到，为了使Ruby代码更快，只需要考虑三件事：

①通过避免额外的分配和内存泄漏来优化内存；

②编写速度更快的迭代器，减少时间和内存；

③通过专业工具完成工作来减少Ruby代码。

这些技术的优点在于，你可以将它们应用于任何Ruby程序，使其速度提高十倍。 但是大多数Ruby开发人员都在编写Rails应用程序，所以现在是时候深入了解并将我们的优化技术应用于Ruby on Rails。



**使Rails更快**

原则上，你已经知道如何使Rails更快：我们在前一章中讨论过的相同性能优化策略适用于任何Rails应用程序。使用更少的内存，避免在迭代器中进行繁重的函数调用，并减少Ruby和Rails的编写。这些是使Rails应用程序更快的重要因素，您将在本章中学习如何应用它们。

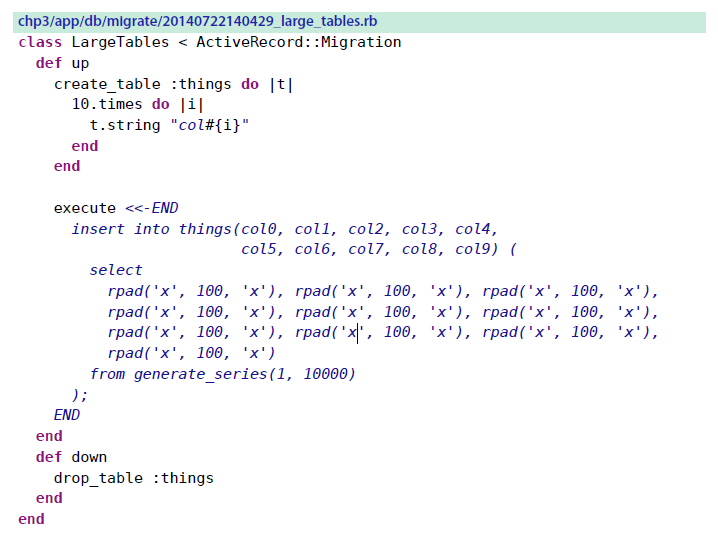
但在我们开始之前，请确保你至少已经设置并运行了裸Rails应用程序。 你将在本章中看到的所有示例，都需要具有数据库连接的Rails 4.x应用程序。 并且假设我们都在使用PostgreSQL 9.x数据库。PostgreSQL是我的首选，不仅因为它是性能最好的免费数据库之一。我特意选择它是因为我需要大量的随机数据用于示例，并且使用Postgres特定的generate\_series函数很容易生成。这让我们可以从空数据库开始，并根据需要在迁移中添加模式和数据。

所以，拿Rails应用程序（裸机或你自己的），让我们优化它。

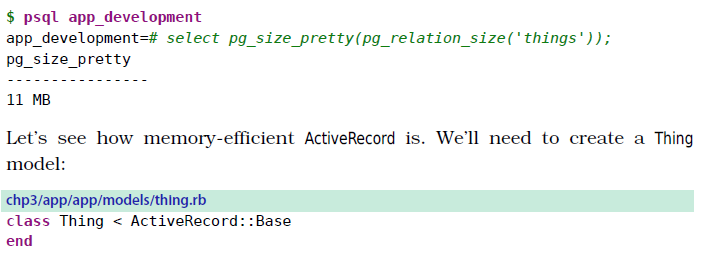
**使ActiveRecord更快**

ActiveRecord是数据的包装器。 根据定义，它会消耗内存，它确实是这样。 事实证明，在数据对象和原始内存中，开销非常大。

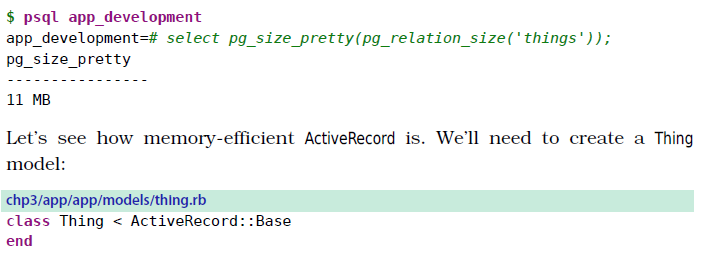
要查看开销，让我们创建一个包含10个字符串列的数据库表，并用10,000行填充它，每行包含10个100个字符串。（代码从下一页开始。）



此迁移创建了1000万字节的数据（10,000 \* 10 \* 100），大约9.5 MB。 数据库在存储时非常有效。 例如，我的PostgreSQL安装仅使用了11 MB：



让我们看看ActiveRecord的内存效率如何。 我们需要创建一个Thing模型：



我们需要将上一章中的wrapper.rb度量助手调整为Rails：

1. 修复常见性能问题

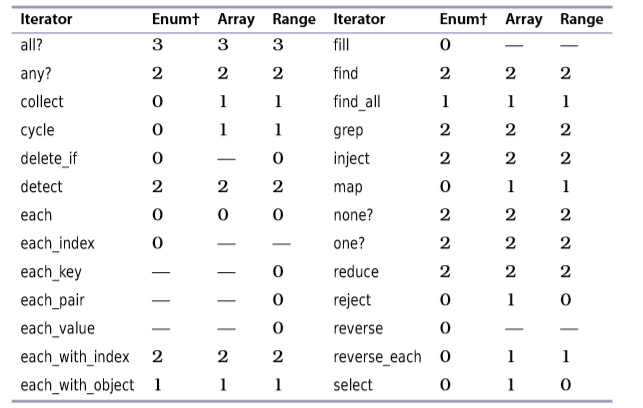
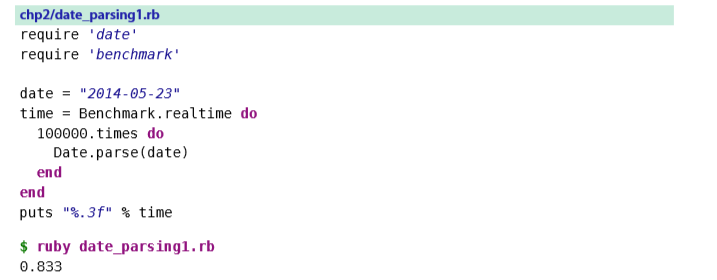


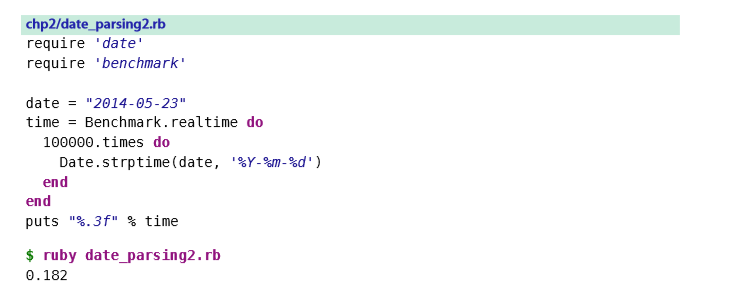
表1—迭代器创建的额外T节点对象数

日期解析：

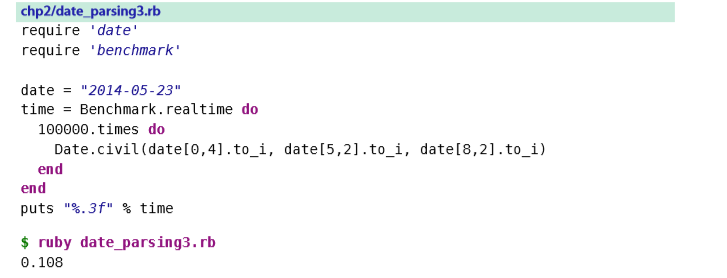
Ruby中的日期解析通常很慢，但是这个函数对性能尤其有害。让我们看看它在循环中使用了多少时间，循环迭代次数为100000次：



每一个日期解析调用只需要0.02毫秒，但在一个相当大的循环中，这就转化为几乎一秒钟的执行时间。更好的解决方案是让日期解析器知道要使用哪种日期格式，如下所示：



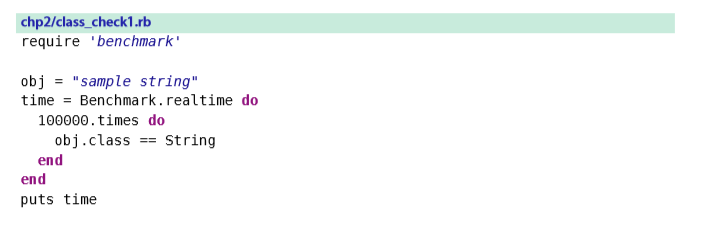
已经快了4.6倍。但是，完全避免日期字符串解析甚至更快：

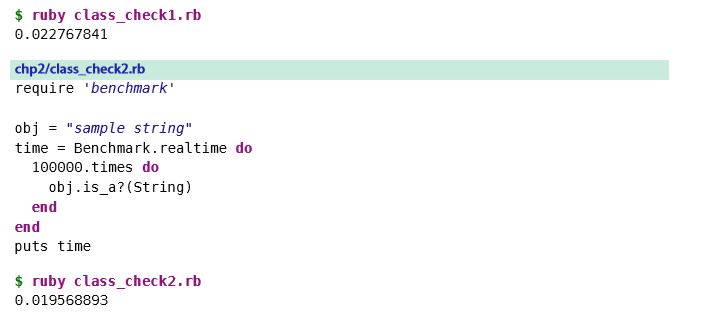


虽然稍显丑陋，但这段代码的速度几乎是原始代码的8倍，比日期strptime版本快了近2倍。

Object#class, Object#is\_a?, Object#kind\_of?

当用于循环或经常使用的函数（如构造函数或==比较运算符）中时，这些方法会产生相当大的性能开销。





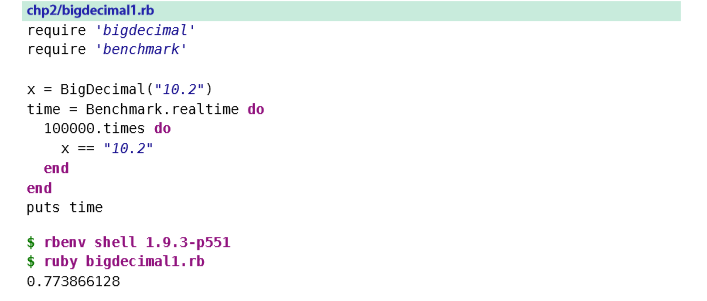
在一个中等大的循环中，同样需要100000次迭代，这样的检查需要19-22毫秒。这听起来并不糟糕，但是，例如，一个Rails应用程序可以对每个请求调用超过100万次的比较运算符，并花费超过200毫秒进行类型检查。

最好将类型检查从迭代器或经常调用的函数和运算符中移开。如果你做不到，不幸的是你做不到什么。

BigDecimal::==(String)

从数据库获取数据的代码经常使用大小数。对于Rails应用程序来说尤其如此。这样的代码通常从它从数据库中读取的字符串中创建一个bigdecimal，然后将其直接与字符串进行比较。

关键是，在Ruby 1.9.3及更低版本中，进行这种比较的自然方法是令人难以置信的缓慢：



选择性加载和预加载属性

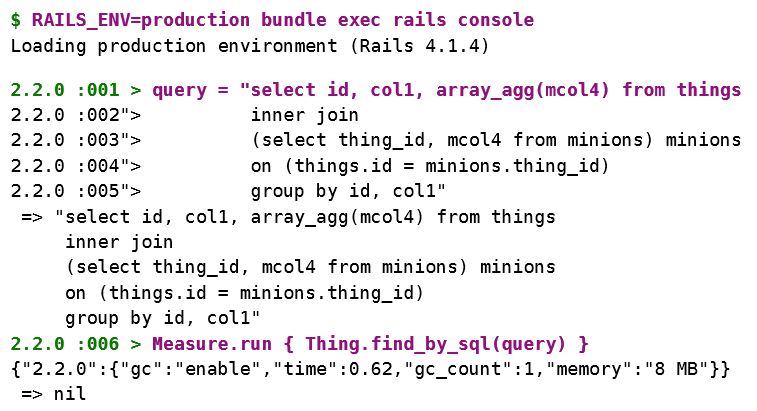
如果从“仅加载您需要的属性”部分获取我的建议，并仅选择我们需要的值，那将会更好。但是有一个问题。Rails没有从依赖模型中选择列子集的便捷方法。例如，这个操作会失败：

Thing.all.includes(:minions).select("col1", "minions.mcol4").load

它失败了，因为includes（：minions）运行了一个额外的查询来获取所选事物的minions。并且Rails不够聪明，无法确定哪些选择值属于Minions表。如果我们从belongs\_to关联的一侧查询，我们需要使用joins。

Minion.where(id: 1).joins(:thing).select("things.col1", "minions.mcol4")

从has\_many侧joins将返回相同Thing对象的重复项，在我们的示例中为10个重复项。为了解决这个问题，我们可以使用PostgreSQL特定的array\_agg功能来聚合连接表中的列数组。



仅仅看一下内存消耗：全部预加载仅需要8MB而不是518MB。最为回报，它运行速度比原来快了20倍。

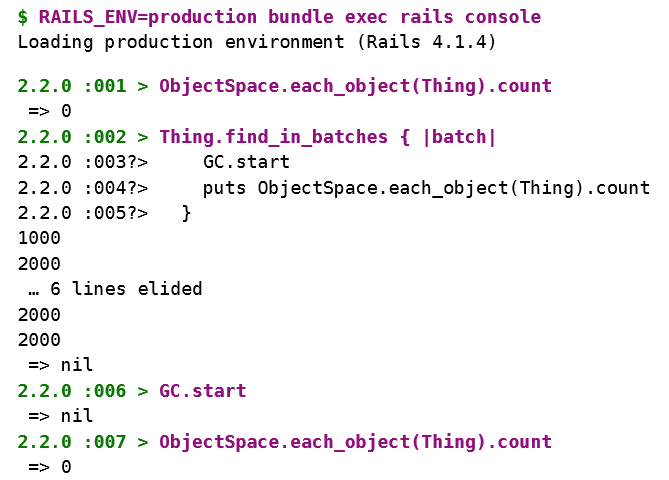
限制你选择的值的数量可以节省几秒的执行时间和数百兆的内存。

使用Each！Rails模式下为find\_each和find\_in\_batches

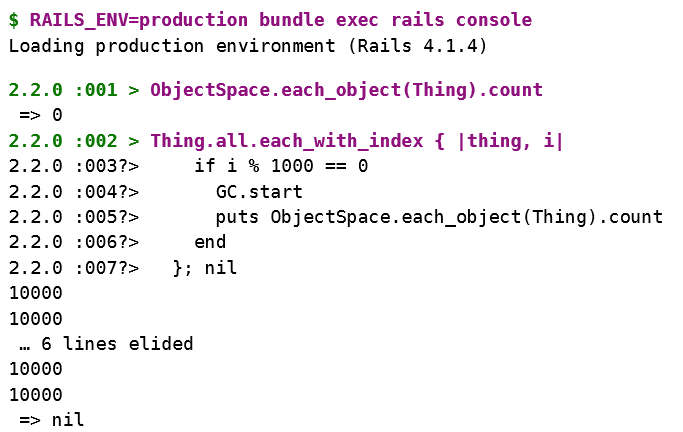
实例化很多ActiveRecord模型是很昂贵的。Rails开发人员了解后添加了两个函数来在大型数据集中批量循环。ind\_each和find\_in\_batches都会默认加载1,000个对象并将它们返回给你——第一个函数，一个接一个。后者为一次性加载整个批次。

您可以使用：batch\_size选项请求更小或更大的批次。

find\_each和find\_in\_batches仍然需要加载内存中的所有对象。那么它们如何提高性能呢？和使用each相同！从27页的Use the Each!模块开始。你一旦完成批处理，GC便可以收集它。让我们看看它是如何工作的。



GC确实从先前的批次中收集对象，因此在迭代期间内存中不会超过两个批次。将此与Thing.all返回的对象列表中的常规的each迭代器进行比较。



在这里，我们在每个循环的整个持续时间内保留10,000个对象。这会增加总内存消耗和GC时间。如果数据集太大，它还会增加内存不足的风险（请记住，ActiveRecord需要3.5倍的空间来存储数据）。

在没有实例化模型的情况下使用ActiveRecord

如果你只需要运行数据库查询或更新表中的值，请考虑使用以下不实例化模型的ActiveRecord函数。

* ActiveRecord::Base.connection.execute("select\*fromthings")

此函数执行查询并返回其未解析的结果。

* ActiveRecord::Base.connection.select\_values("selectcol5fromthings")

与上一个函数类似，但仅从查询结果的第一列返回值数组。

* Thing.all.pluck(:col1,:col5)

与前两个函数不同。该函数返回一个值数组，其中包含你在pluzer参数中指定的整行或列的值。

* Thing.where("id<10").update\_all(col1:'something')

更新表中的列值。

这些操作不仅可以节省内存，而且使运行速度更快，因为它们既不实例化模型，也不在过滤器之前/之后执行。他们所做的只是运行纯SQL查询，在某些情况下，返回数组。

使ActionView更快

模板渲染花费的时间比控制器代码长，这并不罕见。但是你可能认为你不能做太多事情来使它加速。大多数模板只是对渲染辅助函数的调用的集合，这些函数是你没有编写的，并且无法真正被优化 - 除非它们在循环中被调用。

渲染基本上是字符串操作。我们已经知道，这需要占用CPU时间和内存。在循环中的速度需要对每次已经很慢的迭代进行乘方。因此，每次迭代模板中的大型数据集时，请查看是否可以对其进行优化。

我们在每一个循环的整个过程中保留10,000个对象。这增加了总内存的消耗和GC的时间。它也增加了如果数据集太大，内存会耗尽的风险（记住，活动记录需要3.5倍的空间来储存数据）。

**在不示例模型的情况下，用查询**

如果你所需要的只是运行数据库查询或更新表中的列，考虑使用以下不实例化的查询函数模型模型。

* 查询：:Base.connection.execute("select \* from things")

这个函数执行查询并返回未解析的结果。

* 查询：:Base.connection.select\_values("select col5 from things")

类似于前一个函数，但是它的返回值只返回数组查询结果的第一列

* Thing.all.pluck(:col1, :col5)

前两个函数的变化，返回一个包含要提取的参数中指定的整行货列的数组的值

* Thing.where("id < 10").update\_all(col1: 'something')

更新表中的列。

它们不仅可节省内存，而且运行速度更快，因为它们都不能实例化模型或者再过滤器之前/之后执行。他们所有能做的是简单地跑SQL查询，在某些情况下，返回的数组作为结果。。

**让 ActionView 更快**

模板呈现花费的时间比控制器代码长并不罕见。但是你可能认为你不能做什么来加快它。大多数模板只是一个调用集合，用来呈现您没有编写的helper函数。而且它不能真正优化，除非在循环中调用它们。

呈现基本上是一个字符串的操作。我们已经知道了，这需要CPU时间和内存。在一个循环中，我们把已经存在的很慢的效果相乘。因此，每次在模板中迭代大型数据集时，请查看是否你可以优化它。

Rails模板呈现具有与Ruby想类似的性能特征迭代器。它可以做任何事情，直到你在一个循环里渲染局部模板。对于这个，有两个原因。

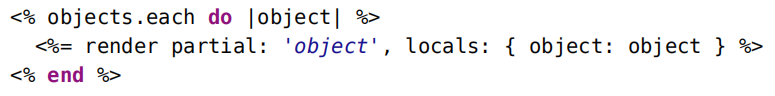
首先，渲染是有代价的。初始化视图对象需要时间，计算执行上下文，并传递所需要的变量。所以，每一个循环中呈现的部分应该是性能不佳的首要原因。

其次，大多数Rails的视图助手都是不安全的迭代器。一个电话不会让你慢下来，但是成千上万个就会让你慢下来。

这两个潜在的性能问题实际上是相同的，在“避免创建其他对象的迭代器”疑问中已经讨论过，在第27页上注意迭代器不安全的Ruby标准函数，在第29页上只需要更高一级的抽象和我们只适用于Rails。所以，详细讨论这些问题，看看我们可以优化什么。

**在循环中更快地呈现局部**

当被要求呈现一组对象时，你的模板代码可能会这样做，看起来是这样的：



这段代码本身没有什么问题，只是在很大的范围内变慢了对象的集合。有多慢呢？我在不同版本的Rails中测量了10,000个空的局部模板，结果并不令人满意。



表2-渲染10000个局部模板

禁用GC后测量，以秒为单位。结果与分位数成线性关系。

虽然10,000个对象不是一个大数据集，但是只呈现占位符，因为使用最新的Rails，他们会让您后退两秒。这是令人不安的。同样令人不安的是，每一个后续版本的Rails的渲染也会变得越来越糟糕。但在你陷入对于Rails 2.x的美好回忆之前，我要指出，即使是0.3秒什么也不做也已经是0.3秒了。Rails 3.0及更高的版本针对这个问题提供了一个名为render collection的解决方案。



或者，你也可以选择更简短的表示方法：

<%= render @objects %>

这会为集合的每个成员插入一个部分，自动计算部分名称并传递局部变量。[[1]](#footnote-1)这样操作能够快20倍。

|  |  |
| --- | --- |
| Rails 3.x | Rails 4.x |
| 0.066 ± 0.001 | 0.100 ± 0.005 |

**表3—渲染10000个对象的集合用时**

渲染集合的速度快的原因是它只是初始化模板一次。然后它重用相同的模板来渲染集合中的所有对象。在循环中渲染10000个部分将不得不重复初始化10000次。

初始化模板需要做多少工作？我已经在Rails 4中分析了10000个部分的渲染来说明这一点。我们来看看摘要。

|  |  |
| --- | --- |
| 操作 | 占总时长百分比 |
| 日志记录 | 45% |
| 查找和读取模板（从磁盘或缓存） | 21% |
| 设置执行上下文（局部变量等） | 9% |
| 模板类实例化 | 5% |
| 渲染 | 5% |
| 其它 | 15% |

我想我们都会对此感到惊喜。实际渲染只需要5％的时间。难怪如果我们跳过初始化，我们将获得两个数量级的加速—正如我们的测量结果一样。

让我们看看为什么日志记录占用了45％的时间。事实证明，在生产模式下使用默认的config.log\_level =:info，Rails会产生过多的输出。

INFO --: Started GET "/test" for 127.0.0.1 at 2014-08-13 10:21:40 -0500

INFO --: Processing by TestController#index as HTML

INFO --: Rendered test/\_object.html.erb (0.1ms)

*«9998 more object.html.erb partial rendering notifications»*

INFO --: Rendered test/\_object.html.erb (0.0ms)

INFO --: Rendered test/\_dummy.html.erb (1904.0ms)

INFO --: Rendered test/index.html.erb within layouts/application (1945.4ms)

INFO --: Completed 200 OK in 1952ms (Views: 1948.6ms | ActiveRecord: 0.0ms)



很可能会渲染10000个部分的Rails应用程序不是很多。但是1000个似乎不是一个无法达到的数字。我们来做一些算术。想象一下，你在循环中渲染了100个具有部分的对象。现在想象部分调用10个其他部分。这些数字看似合法。如果你渲染了一个包含10列的分页表，你将获得这样的设置。我们有多少渲染部分调用？已经有1000个了。我们在渲染部分功能中花多少时间？根据我的测量结果，大约200毫秒。如果我们考虑实际渲染有用内容的时间，我们将轻松超过1秒的响应时间标记，而这对任何Web应用程序来说都是不可接受的。

**乔的提问：**

**但Rails应用程序很少需要渲染10,000个部分，不是吗？**

您可能不希望使用config.log\_level =:warn完全使您的日志静音，但这样做会带给您两倍的加速。

|  |  |
| --- | --- |
| config.log\_level = :info | config.log\_level = :warn |
| 1.840 ± 0.045 | 0.830 ± 0.049 |

**表4—渲染具有不同日志级别的10000个对象的集合用时**

这仍然没有渲染集合那么快（0.1秒）。剩下的0.7秒去哪儿了？事实证明，Rails使用Observer模式实现了一个记录器。部分渲染会触发render\_partial.action\_view事件。当发生这种情况时，ActionSupport::LogSubscriber会收到通知，然后运行Logger来生成输出。这个管道代码大约需要0.2秒。剩余的步骤是模板初始化和执行上下文评估。

渲染集合没有任何开销，也不会产生过多的日志输出。这使得它明显优于在循环中渲染部分。

Rails 2.x中没有渲染集合。但是，如果您仍在使用该版本，请尝试使用模板内联插件。[[2]](#footnote-2)通过在Rails编译之前将部分代码文本插入父模板，可以实现相同的效果。

这是我在Acunote[[3]](#footnote-3)工作时写的，Acunote是使用Ruby on Rails构建的在线项目管理系统。我们在页面上渲染了数百个任务，每个任务有8-10个字段。对于每个字段，我们有一个单独的部分用于渲染，并且在Rails 2.x中没有渲染集合。

要使用它，请将插件添加到您的Rails应用程序中，并将内联：true添加到呈现语句：

<% @objects.each do |object| %>

<%= render partial: 'object' , locals: { object: object }, inline: true %>

<% end %>

Rails从未看到呈现部分调用，因此我们得到了相同的两个数量级性能改善的结果。

Rails 2.x Rails 2.x with template inliner

0.003 ± 0.0001 0.335 ± 0.006

表5-内联呈现10，000个部分的时间

用禁用GC测量，以秒为单位。结果与分词数成线性关系。

避免Iterator不安全的帮手和函数

所有呈现帮助程序都是我所说的迭代器不安全的。它们需要时间和记忆，所以在循环中使用时要小心，尤其是link\_to, url\_for,和img\_tag。

出于两个原因，我没有比小心更好的建议了。首先，您不能避免使用这些助手(特别是在较新的Rails中)。其次，很难对它们进行基准测试。帮手的性能取决于太多的因素，使任何综合基准无用。例如，当路由的复杂性增加时link\_to和url\_for会变慢。当您添加更多的资产时，img\_tag的性能更差。在一个应用程序中，在循环中呈现1000个URL是安全的，而在另一个应用程序中则不是。所以…请注意。

Takeaways

为了更快地使用Rails应用程序，我们在第2章“修复常见性能问题”第13页中学到了同样的技术：

* 优化ActiveRecord占用的内存，通过积极的预加载、选择性属性获取和批量数据处理。
* 将视图中的显式迭代器替换为呈现集合，这将减少时间和内存。
* 让您的数据库服务器执行数据操作。

这一章和前两章包含了我可以给您的所有建议，以使您的Ruby和Rails代码更快。继续，应用它，并收获你的绩效利益。

那么，你可能会问，为什么这不是书的结尾呢？因为任何菜谱式的建议都只对众所周知的情况有好处。当性能因为一个完全不同的原因而退化时，它会让你措手不及，这是我没有经历过的，也是我无法预测的。

如果这些建议都不起作用，现在该学会做些什么了。当这种情况发生时，您需要分析应用程序，了解什么可能出错，优化，度量，并确保减速不再发生。在接下来的三章中，你将学习如何做到这一点。

第四章

分析

好的，所以你已经学会了关键的性能优化技术，并可以将它们应用到你的代码中。但是，当我们讨论过的技术都不起作用时你会怎么做？

你要分析。

分析是回答“什么减慢了代码的速度？”唯一可靠方法。分析可能困难又耗时，但他确实没有任何捷径。如果你不能通过查看代码或采取有根据的猜测来优化，你就必须进行分析。

一旦你确切地知道什么在减慢你的速度，修复它就变得不值一提了。所以现在我将教你分析的秘密，这能使你更容易地发现是什么变慢了。

让我们通过将其分解为两个基本部分来开始我们对分析的探索。首先，测量内存或CPU使用率并将其归因于代码中的特定位置，通常是函数调用。其次，解释结果以识别代码的缓慢部分。这是两种截然不同的活动，你需要以不同的方式思考它们。

测量是一项纯粹的工程任务，而且很简单。你可以手动完成或使用分析器工具。我将展示如何使用这些工具。

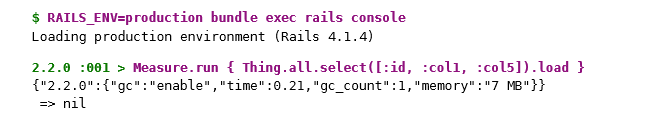
解释测量更复杂，但秘诀是将其视为工艺，而不是工程任务。我见过许多优秀的软件开发人员正在放弃分析，正是因为他们试图像工程师一样地去分析。你的左脑会认为分析是麻烦又无趣的。但是当让你的右脑参与其中后，相反你会发现它有趣且令人兴奋。

因为分析是一种工艺，至少我会把它当作一种工艺来教——换句话说，通过例子。我将展示我如何使用具体示例来分析我的代码，然后你可以通过实例来提炼出适合你的技术。一旦你学会示例中的模式，这将变得很容易。所以，动用你的右脑，让我们开始吧。

ActiveRecord很方便的，但是ActiveRecord提供的方便代价高昂。我知道我不能说服你让你停止使用ActiveRecord。但是你需要了解使用它的后果。在80%的情况下，开发速度比执行速度的花费更有价值。在剩下的20%的情况下，你有其他的选择。下面让我演示你看看：

**只加载你所需的属性**

你的第一选择就是只加载你要用的数据。Rails将使得这很简单。像下面这行代码这样：

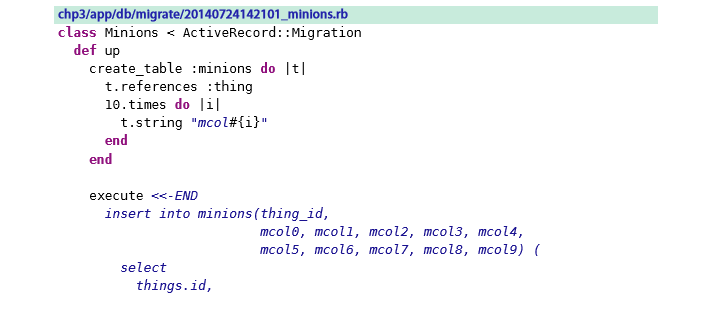


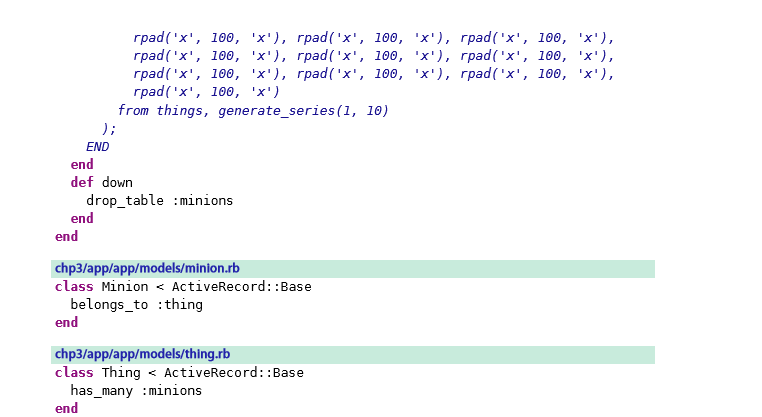
它占用的内存比Thing.all.load少5倍，运行速度比Thing.all.load.快1.5倍。你写的代码的行数越多，在查询中添加select就越有意义，特别是在连接表时。

**积极预载**

另一个最佳实践是预载。每次查询has\_many表或belongs\_to表的关系时，都要预加载。

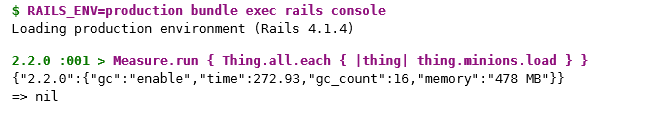
例如，让我们向我们的Thing中添加一个has\_many关系调用。我们需要建立迁移和ActiveRecord模型。





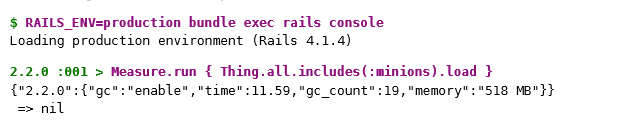
利用RAILS\_ENV=productionbundleexecrakedb:migrate运行迁移，你就会在数据库中的每一个Thing中得到10个minions。

在没有预加载的情况下遍历该数据不是一个好主意。



希望您能幸运的等到这一行运行代码完成。它不仅需要将所有内容加载到内存中，还需要对数据库执行10,000条查询，以便得到每一个Thing的Minssions。

预载是一个更好的办法。



由于Rails版本的不同，这可能稍会微降低内存的效率。但是代码运行的的速度快了25倍，因为Rails只执行两个数据库的查询——一个加载things，另一个加载minions。

3.81 0.000 0.000 0.000 0.000 1 String#gsub!

1.22 0.000 0.000 0.000 0.000 1 MatchData#begin

0.67 0.000 0.000 0.000 0.000 1 String#[]=

0.38 0.000 0.000 0.000 0.000 1 Fixnum#div

0.25 0.000 0.000 0.000 0.000 1 MatchData#end

表示递归调用的方法

我们的代码花费的时间超过我传递给正则表达式的日期的60％。大多数剩余时间由Date＃parse函数本身占用，可能用于Date实例化。

现在让我们看看如何使用命令行工具。

2. ruby-prof Command-Line Tool，2. ruby​​-prof命令行工具

您不必向程序本身添加任何检测以使用命令行工具。所以程序可以简单地这样：

chp4/ruby\_prof\_example\_command.rb

require *'date'*

GC.disable

Date.parse(*"2014-07-01"*)

运行ruby-prof命令对其进行分析。

**$ ruby-prof -p flat -m 1 -f ruby\_prof\_example\_command\_profile.txt\**

**ruby\_prof\_example\_command.rb**

这里-p选项告诉使用哪台打印机输出报告（在我们的例子中再次使用平面打印机）。并且-m通过抑制执行时间少于指定时间百分比的所有函数来限制输出。在这种情况下，我不关心代码花费时间少于1％的地方。

你应该得到这样的个人资料：

chp4/ruby\_prof\_example\_command\_profile.txt

Thread ID: 69883126035220

Fiber ID: 69883132260680

Total: 0.002094

Sort by: self\_time

%self total self wait child calls name

17.37 0.001 0.000 0.000 0.001 3 \*Kernel#gem\_original\_requi

16.97 0.000 0.000 0.000 0.000 2 Regexp#match

10.85 0.001 0.000 0.000 0.000 1 <Class::Date>#parse

2.36 0.000 0.000 0.000 0.000 113 Module#method\_added

1.83 0.000 0.000 0.000 0.000 6 IO#set\_encoding

1.57 0.000 0.000 0.000 0.000 1 String#gsub!

1.23 0.000 0.000 0.000 0.000 6 MonitorMixin#mon\_enter

1. http://guides.rubyonrails.org/layouts\_and\_rendering.html#rendering-collections [↑](#footnote-ref-1)
2. http://github.com/acunote/template\_inliner [↑](#footnote-ref-2)
3. http://www.acunote.com/ [↑](#footnote-ref-3)