说说 Rails 的套娃缓存机制

徐峥

彩程设计软件工程师

已关注

13 人赞了该文章

Rails 4.0 以后，开始推广一种称为「俄罗斯套娃」的缓存机制，这是一种使用 Fragment Caching 技术的缓存机制，在数据库做完查询以后，如果记录没有变化，那么对应的页面不会被 Rails 重新渲染，而是直接从缓存里取出，拼装好以后，返回给客户。

Tower 正是借鉴了这套缓存机制，给那些访问 Tower 的用户提供流畅的使用体验，今天跟大家分享一下我们的经验，和一些需要注意的坑。

1. 套娃是怎么套的

拿 Tower 的项目详情页为例，我们可以把这个页面明显的分成一个个的 Section，比如「讨论区」、「任务清单区」、「文件区」、「文档区」、「日历事件区」，我们可以把每一个列表区域设置为一个独立的缓存，这样，如果列表的数据没有更新的话，在渲染项目详情页的时候，就可以直接从缓存里读取之前生成好的数据。

Russian Doll

成员 设置 进展 统计

说说 Rails 的套娃缓存机制

讨论 发起新的讨论

Section Cache: 讨论

	古灵	这是讨论A R.T.	3分钟前
	古灵	这是讨论B R.T.	3分钟前
	古灵	这是讨论C R.T.	3分钟前

全部3条讨论

任务清单 添加清单

Section Cache: 任务清单

任务清单A

- 任务 A-1 未指派
- 任务 A-2 未指派
- 任务 A-3 未指派

添加新任务

任务清单B

- 任务 B-1 未指派
- 任务 B-2 未指派
- 任务 B-3 未指派

添加新任务

任务清单C

- 任务 C-1 未指派
- 任务 C-2 未指派
- 任务 C-3 未指派

任务筛选

所有成员

今天

← 筛选结果已用 荧光笔 标记

文件 上传文件

Section Cache: 文件


参考资料


需求文档


设计图

如何整理项目文件.pdf

查看全部文件 查看附件流

文档 创建新文档

Section Cache: 文档

在线文档A

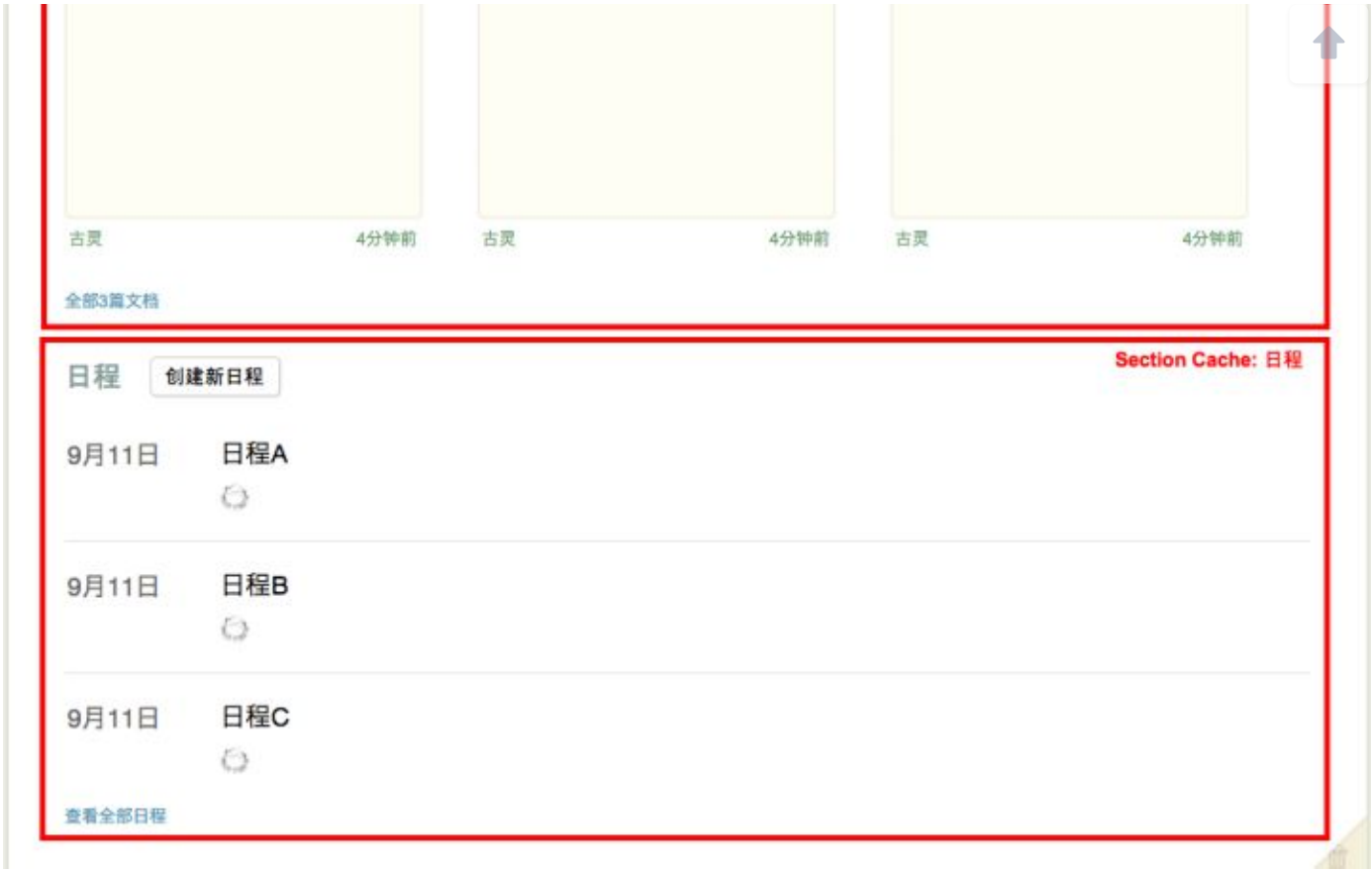
这是文档A

在线文档B

这是文档B

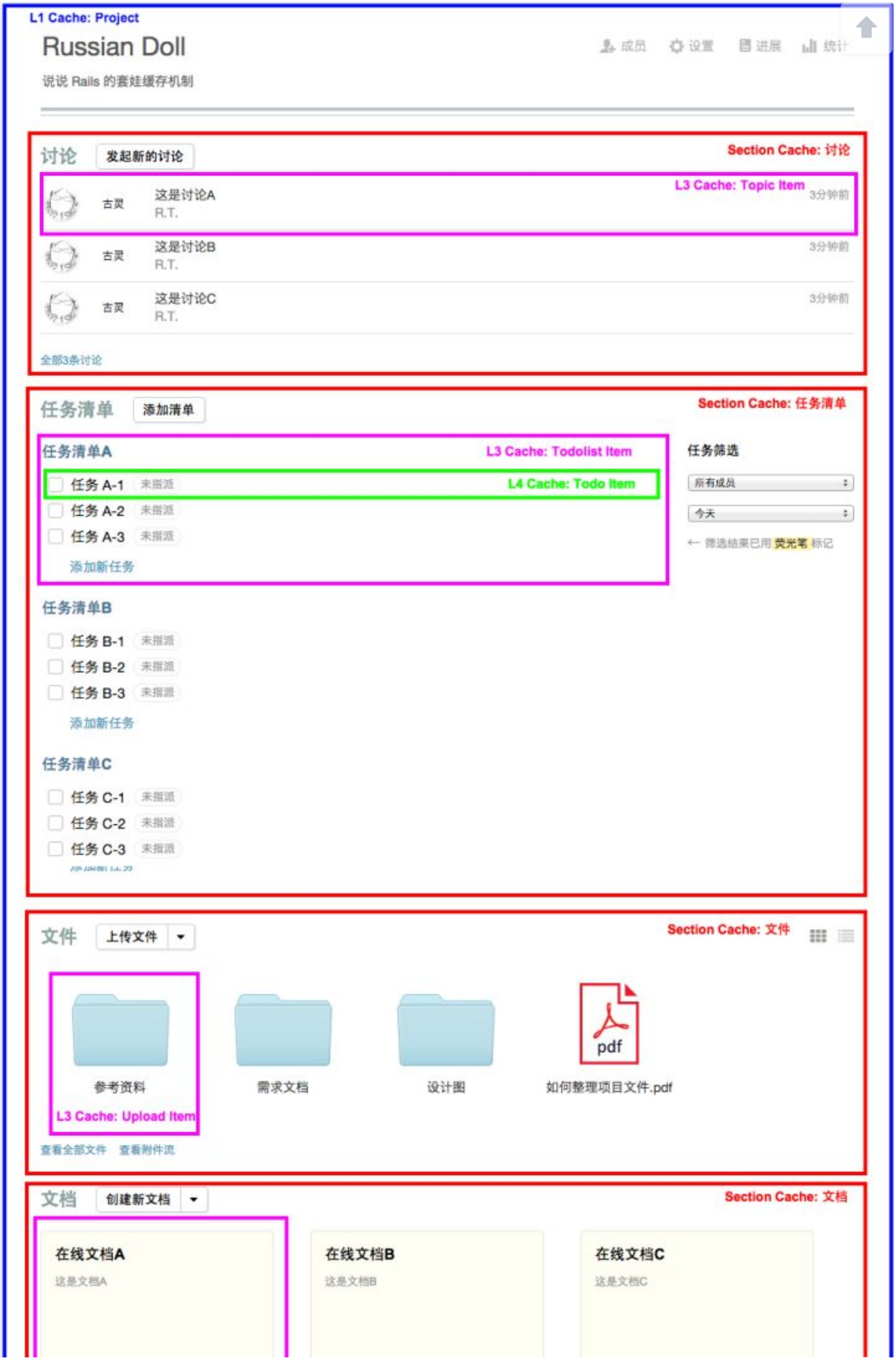
在线文档C

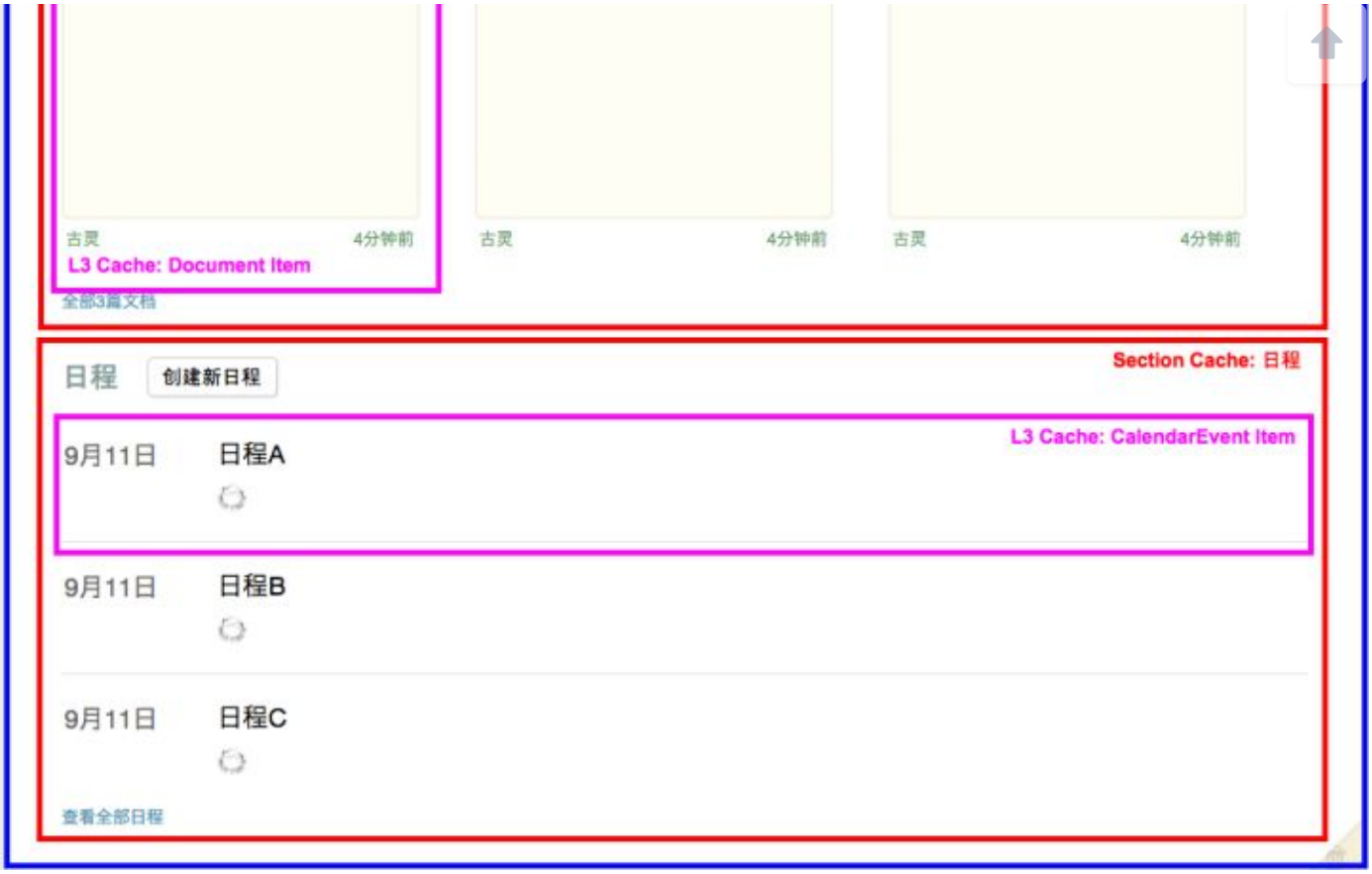
这是文档C



那么，所谓的套娃在哪儿呢？实际上，除了可以把上面的 Section 放到缓存里，我们也可以把整个项目详情页放入缓存，这样，如果某一个项目里的数据没有任何更新，访问这个详情页就可以直接读取详情页的缓存，连下面的 Section 缓存都不用碰了。

同样，对于某一个列表缓存，比如「讨论区」，我们可以看到里面会放三条讨论 item，这里其实每一个 item 也可以对应放入一个独立的缓存，这样如果只有其中一条 item 有更新的话，其它两条数据是不会被重新渲染，而是直接从缓存区读取的。这样分拆下去，我们大概可以把整个项目详情页的缓存弄成下面这个模样：


L1 Cache: Project
Russian Doll
成员
设置
进展
统计
说说 Rails 的套娃缓存机制
讨论
发起新的讨论
Section Cache: 讨论
L3 Cache: Topic Item
古灵
这是讨论A
R.T.
3分钟前
古灵
这是讨论B
R.T.
3分钟前
古灵
这是讨论C
R.T.
3分钟前
全部3条讨论
任务清单
添加清单
Section Cache: 任务清单
任务清单A
L3 Cache: Todolist Item
任务 A-1 未指派
L4 Cache: Todo Item
任务 A-2 未指派
任务 A-3 未指派
添加新任务
任务筛选
所有成员
今天
← 筛选结果已用 荧光笔 标记
任务清单B
任务 B-1 未指派
任务 B-2 未指派
任务 B-3 未指派
添加新任务
任务清单C
任务 C-1 未指派
任务 C-2 未指派
任务 C-3 未指派
文件
上传文件
Section Cache: 文件
参考资料
L3 Cache: Upload Item
需求文档
设计图
pdf
如何整理项目文件.pdf
查看全部文件
查看附件流
文档
创建新文档
Section Cache: 文档
在线文档A
这是文档A
在线文档B
这是文档B
在线文档C
这是文档C



这样不就一层一层的套起来了么？

接下来我们看看，如果这个页面的某一条数据，比如「任务 A-1」的内容被改变了，会发生什么。

首先，这条任务自己对应的 L4 Todo Item 缓存失效了，所以在拼装外面的 L3 级「任务清单A」缓存的时候，会从缓存里获取任务 A-2、A-3 的缓存，速度嗖嗖快，快到可以忽略不计，然后对任务 A-1 重新渲染一次，放入缓存，这样「任务清单A」通过直接从缓存里读取两条任务（A-2 和 A-3），以及渲染一条新的（ A-1 ）生成了整个 L3 Todolist Item 的页面片段。剩下的「任务清单B」和「任务清单C」，都没有变化，因此由在生成「任务清单」Section 缓存的时候，直接拼装即可。

其它几个 Section 片段因为和任务没有任何关系，所有缓存都不会过期，因此这几个 Section 的页面片段都是直接从缓存里捞出来，同样嗖嗖快。

最后，整个项目详情页把这几个 Section 拼装起来，返回给客户。从上面的过程可以看出，只有「任务 A-1」这个片段的页面被重新渲染了。

所以，这种套娃式的缓存，能够保证页面缓存利用率的最大化，任何数据的更新，只会导致某一个片段的缓存失效，这样在组装完整页面的时候，由于大量的页面片段都是直接从缓存里读取，所以页面生成的时间开销就很小。

那么，套娃是如何在缓存中存取页面片段的呢？主要是靠一个叫做 cache_key 的东西来决定的。

2. cache_key

我们在页面上可以使用一个叫做 cache 的方法，把一坨 HTML 代码片段放在一个 Fragment Cache 里，以项目详情页为例，我们的代码可能是这个样子的：

```
<!-- views/projects/show.html.erb -->
<% cache @project do %>
  <%= @project.name %>

  <!-- Section Topics -->
  <% cache @top3_topics.max(&:updated_at) do %>
    <%= render partial: 'topics/topic', collection: @top3_topics %>
  <% end %>

  <!-- Section Todolists -->
  <% cache @todolists.max(&:updated_at) do %>
    <%= render partial: 'todolists/todolist', collection: @todolists %>
  <% end %>

  <!-- Section Uploads -->
  <% cache @uploads.max(&:updated_at) do %>
    <%= render partial: 'uploads/upload', collection: @uploads %>
  <% end %>

  <!-- Section Documents -->
  <% cache @documents.max(&:updated_at) do %>
    <%= render partial: 'documents/document', collection: @documents %>
  <% end %>

  <!-- Section CalendarEvents -->
  <% cache @calendar_events.max(&:updated_at) do %>
    <%= render partial: 'calendar_events/calendar_event', collection: @calendar_events %>
  <% end %>

<% end %>
```

可以看到，在整个页面的最外面，有个最大的「套娃」：<% cache @project %>，这个 cache 使用 @project 作为方法参数，在 cache 方法内部，会把这个对象进行一番处理，最后生成一个字符串，大概是这个样子：

「*views/projects/1-20140906112338*」

这就是所谓的 cache_key，Rails 会使用这串字符串作为 key，对应的页面片段作为内容，存储进缓存系统里。每次渲染页面的时候，Rails 会根据 cache 里的元素计算出对应的 cache_key，然后拿着这个 cache_key 到缓存里去找对应的内容，如果有，则直接从缓存里取出，如果没有，则渲染 cache 里的 HTML 代码片段，并且把内容存储进缓存里。

对于一个具体的 Model 对象，cache_key 的生成机制简单来说，就是：*对象对应的模型名称/对象数据库ID-对象的最后更新时间*。

这里我们能够很容易分析出，一个缓存判断最后是否过期，其实很大程度上只和数据最后更新时间有关，因为在系统里，数据对象对应的模型名称是不变的，对象在数据库里的 ID 一般也是不变的，唯一可能变化的就是最后更新时间。Rails 在创建模型数据表的时候，一般会创建两个默认的 datetime 类型字段，一个是 created_at，一个是 updated_at，而后者正是用来生成 cache_key 的最后更新时间。而且这个时间一般来说不需要我们手动更新，我们都知道如果对一个模型对象调用

save 方法，Rails 会自动帮我们更新这个 updated_at 字段，这样，如果我修改了项目名称，项目的 updated_at 会发生变化，自然的，页面上项目对应的 cache_key 也就会发生变化，因此我们的 <%cache @project %> 也就自动过期了。

继续图3里的示例代码，接下来我们看看第二级套娃：各个 Section 缓存。

拿这个 <% cache @top3_topics.max(&:updated_at) %> 为例，它比 <% cache @project %> 稍微复杂了点。我们首先应该知道的是，@top3_topics 存储的是对应项目里最新创建的三条讨论，这里比较奇怪的是，我们为什么要用一个 max(&:updated_at) 方法呢？

如果我们直接把 @top3_topics 对象作为 cache 的参数 <% cache @top3_topics %>，得到的 cache_key 实际上会是这样的形式：

「views/topics/3-20140906112338/topics/2-20140906102338/topics/3-20140906092338」

看的出来，是每个对象的 cache_key 的组合，我们并不太希望 cache_key 变得这么复杂，特别是当列表元素超过 3 个，比如说有 20 条记录的时候，所以最简单的办法，是取这组数据里最新一个被更新的数据的 updated_at 时间戳，这样生成的 cache_key 就是下面的样子了：

「views/20140906112338」

但是注意，这里有一个问题，就是假如 @top3_topics 一条数据都没有，会出现什么情况？比如我新建的项目，里面理所当然的一条讨论都没有，这个时候，实际上 cache 的是一个空的 relation，对这个空对象调用 max(&:updated) 方法，返回的值永远都是 nil，所以实际上我们是对 nil 进行 cache，不幸的是，所有 nil 的 cache_key 都一模一样，导致这样的缓存片根本不可用，你不知道究竟是对什么数据进行的缓存。另外，加入任务清单 Section 和讨论 Section 最后更新的那条数据的 updated_at 时间戳恰好一样，也会造成两个缓存片混淆的问题。

而解决这个问题的方法很简单，就是给 cache 参数里增加一个特定的字符串标识，比如把 <% cache @top3_topics.max(&:updated_at) %> 改成 <% cache [:topics, @top3_topics.max(&:updated_at)] %>，这样一来，如果 @top3_topics 里一条数据都没有，生成的 cache_key 是这样的：

「views/topics/20140906112338」

带上了「topics」自己的标识，这样就能和其它 nil 类型的缓存区分开了。修改后的项目详情页代码片段如下：

```erb
<!-- views/projects/show.html.erb -->
<% cache @project do %>
  <%= @project.name %>

  <!-- Section Topics -->
  <% cache [:topics, @top3_topics.max(&:updated_at)] do %>
    <%= render partial: 'topics/topic', collection: @top3_topics %>
  <% end %>

  <!-- Section Todolists -->
  <% cache [:todolists, @todolists.max(&:updated_at)] do %>
    <%= render partial: 'todolists/todolist', collection: @todolists %>
  <% end %>

  <!-- Section Uploads -->
  <% cache [:uploads, @uploads.max(&:updated_at)] do %>
    <%= render partial: 'uploads/upload', collection: @uploads %>
  <% end %>

  <!-- Section Documents -->
  <% cache [:documents, @documents.max(&:updated_at)] do %>
    <%= render partial: 'documents/document', collection: @documents %>
  <% end %>

  <!-- Section CalendarEvents -->
  <% cache [:calendar_events, @calendar_events.max(&:updated_at)] do %>
    <%= render partial: 'calendar_events/calendar_event', collection: @calendar_events %>
  <% end %>

<% end %>
```

3. Touch!

我们回过头来再看看套娃缓存的读取机制，访问项目详情页的时候，首先读取最外层的大套娃 <% cache @project %> ，如果这个缓存片对应的 cache_key 在缓存里能找到，则直接取出来并且返回，如果缓存过期，则读取第二级套娃 — 几个列表 Section 缓存，这些缓存根据列表里最新一条数据的更新时间生成 cache_key，如果最新一条数据的更新时间没有变化，则缓存不过期，直接取出来供页面拼装用，如果缓存过期，则继续读取各自的第三级套娃。

等等，这里有个问题，如果我改变了一条任务的内容，也就是作废了任务 partial 自己的缓存，但是包裹任务的任务清单，以及包裹任务清单的项目都没有变化，这样当页面加载的时候，读取到的第一个大套娃 — <% cache @project %> 都没有更新，会直接返回被缓存了的整个项目详情页，所以根本不会走到渲染更新的任务 partial 那里去。对于这个问题的解决方案，是 Rails 模型层的 touch 机制。

简单的说，我们需要让里面的子套娃在数据更新了以后，touch 一下处在外面的套娃，告诉它，嘿，我更新了，你也得更新才行。我们直接看看这个代码片段：

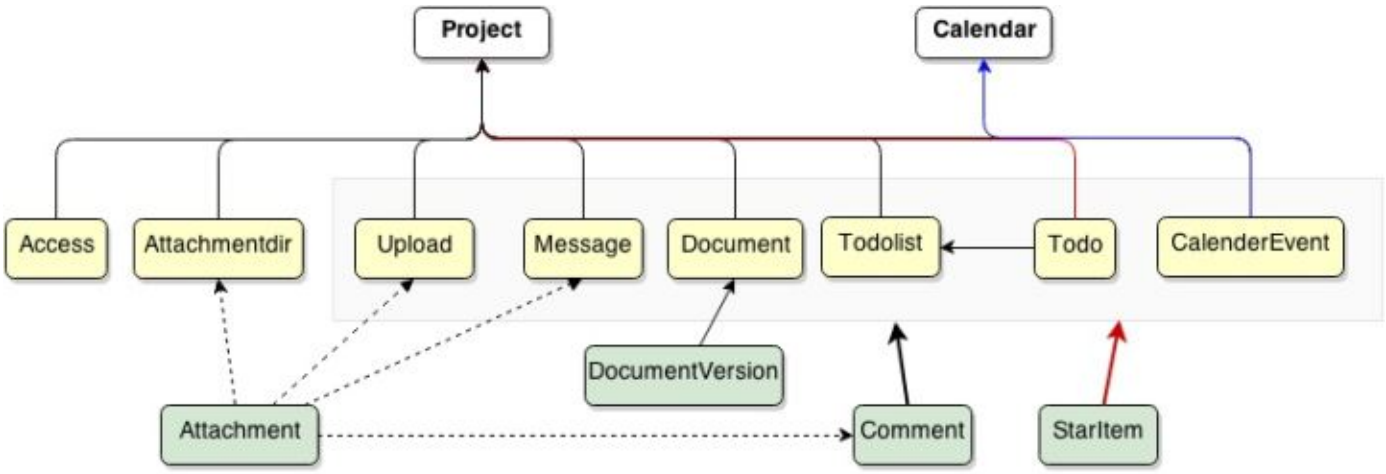
```ruby
class Project < ActiveRecord::Base
  # ....
end

class Todolist < ActiveRecord::Base
  belongs_to :project, touch: true
end

class Todo < ActiveRecord::Base
  belongs_to :todolist, touch: true
end
```

在这里，我们使用 Rails model 的 belongs_to 来声明模型的从属关系，比如一个 Todo 属于一个 Todolist，一个 Todolist 属于一个 Project，而在 belongs_to 后面，我们还传入了一个 touch: true 的参数，这样，当一条 Todo 更新的时候，会自动更新它对应的 Todolist 对象的 updated_at 字段，然后又因为 Todolist 和 Project 之间也有 touch 机制，所以对应 Project 对象的 updated_at 字段也会被更新。放到我们的套娃缓存片里面看的话，就是当一条任务更新以后，「包裹」它的任务清单的缓存片也会被更新，因为对应的 Todolist 对象的 updated_at 时间改变了，而「包裹」这个任务清单的任务清单列表 Section 的缓存片也会失效，因为 @todolists.max(:updated_at) 改变了，接着是「包裹」列表 Section 的项目缓存片过期，因为 @project 对应的 updated_at 也被更新了。

就是通过这么重重 touch 的机制，我们能确保子元素在更新以后，它的父容器的缓存也能过期，整个套娃机制才能正常运作。下面是整个 Tower 里面，各个模型层的 Touch 结构图：



Tower Touch Mechanism

4. 那些踩过的坑

经过上面的介绍，大家应该已经明白了套娃的实际使用方式，看上去很完美不是么？但在我们的实际使用过程中，套娃缓存还是有一些坑需要注意的，这里跟大家分享一下。

我们在开发过程中经常遇到的一个问题，是缓存模板里如果存在「父」元素的情况。我们把 Project 定义为 Todolist 的父元素，把 Todolist 定义为 Todo 的父元素，因为 touch 机制是自底向

上的，从子 touch 到父，但是如果我们的模板是下面这个样子：

```
<!-- Section Todolists -->
<% cache [:todolists, @todolists.max(&:updated_at)] do %>
  <span>清单所属的项目是：<%= @project.name %></span>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

在任务清单模板里，我们需要显示一下项目的名称，也就是一个子元素的模板里，包含了父元素，这个时候如果缓存是 <% cache [:todolists, @todolists.max(&:updated_at) %> 的话，当我们把项目名称修改了，这个缓存片是不会过期的，因此任务清单列表里的项目名也不会改变。

解决这个问题的办法，一是修改任务清单的缓存的 cache_key，改成：

```
<!-- Section Todolists -->
<% cache [:todolists, @project, @todolists.max(&:updated_at)] do %>
  <span>清单所属的项目是：<%= @project.name %></span>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

这样修改项目名称，就能导致缓存片过期，这也是一个普遍的手段，就是把缓存里面存在的所有模型对象统一纳入 cache_key 里面，但是这样存在一个问题，就是因为项目本身是经常被 touch 的，修改任务也会、创建评论也会，所以导致这个任务清单的缓存片会随时失效，缓存命中率降低，所以使用这种方法的时候要仔细考虑，引入父元素作为 cache_key 的一部分，是否会导致这个问题。

另一个办法是，使用实际需要的模型字段来做缓存，比如上面的例子，我们实际上只是需要项目名称，所以可以把缓存改为：

```
<!-- Section Todolists -->
<% cache [:todolists, @project.name, @todolists.max(&:updated_at)] do %>
  <span>清单所属的项目是：<%= @project.name %></span>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

这样只会在项目名称发生改变的时候，更新缓存片，这个方法可能性价比最高，不过如果一个缓存里出现多个模型字段的时候，就要写一串这样的 cache_key，和我们「只对一个具体资源缓存」的原则有些差距，所以一般来说，缓存的具体字段最好不要超过一个。

还有一个处理方法是，在 HTML 结构上做调整，基于我们上面所说的「只对一个具体资源缓存」的原则，这里我们如果针对的是 @todolists 做缓存，那么就应该把其它无关的资源从 HTML 结构里提取出来，比如放到一个外层的 hidden input 里面：

```
<!-- Section Todolists -->
<input type="hidden" name="project_name" value="<%= @project.name %>" />
<% cache [:todolists, @project.name, @todolists.max(&:updated_at)] do %>
  <span id="project-name-span">清单所属的项目是：</span>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

这样可以通过 JS 读取这个属性，再重新注入到模板相应的元素里面。选择这种方案，需要提前根据设计做好规划，把那些需要提取出来的元素放在缓存以外。

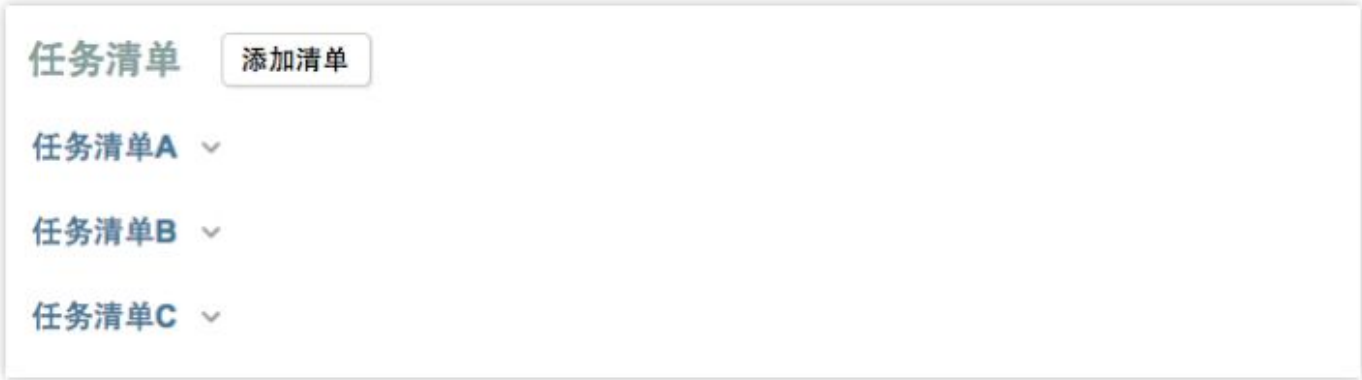
最后还有一个方法，就是不理会它。如果你相信任务清单不会长期不变，而项目名称不会经常变化的话，那么缓存里的项目名称不会随时都是最新版本，就是一个可以被接受的事实了，这需要在产品层面上考虑，我们建议如果遇到这样的问题，不妨先用这种最简单的方式处理，看看用户反馈再决定是否进行调整。

———————————————————— *我是分割线* ————————————————————

我们遇到的第二个问题比第一个问题更加让人头疼，这个问题发生在我们为 Tower 引入一个叫做「访客锁」的新功能的时候。在 Tower 里，用户被分为普通成员、管理员和访客三种，在一个项目里，有些资源比如一条任务清单，是可以设置对访客不可见的，这个在模型层处理起来很简单，只需要增加一个字段来标识一个资源是否是对访客不可见即可，但是一旦和 Fragment Caching 结合的时候，就有问题了。在引入访客锁功能之前，任务清单列表的 Cache 是这样的：

```
<!-- Section Todolists -->
<% cache [:todolists, @todolists.max(&:updated_at)] do %>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

这里 @todolists 是从项目里取出来的所有未完成的任务清单，然后使用 max(&:updated_at) 时间戳来作为 cache_key，这样在一条任务清单更新以后，这个最后更新时间会变化，cache_key 也就变化了。但是在引入访客锁以后，这就会有潜在问题了。假如我们现在有如下图所示的三条任务清单：



我们首先将「任务清单B」加锁，然后再去修改一下「任务清单A」的名称，这个时候整个清单列表的 max(&:updated_at) 时间就是「任务清单A」的 updated_at 时间，如果一个普通成员先打开项目详情页，根据这个更新时间，会缓存一个含有三条任务清单的页面，接着一个访客再打开同一个项目详情页，会出现什么情况呢？这个访客会看到三条同样的任务清单，「任务清单B」加锁是无效的！这是因为对于访客来说，虽然在控制器里查询出来的任务清单只有 A 和 C 两条，但是对于这两条任务清单，最后更新的是 A 的 updated_at 时间戳，这个和能看到三条清单的普通成员以及管理员是一样的，因此他们的任务清单列表的 cache_key 是一样的，取出来的缓存片也一样。

关于这个问题我们考虑了很久，最后发现只有两种解决方案，要么是彻底放弃对这种列表类型的片段做缓存，要么就是遍历列表里的所有子元素，把各自元素的 cache_key 组合起来再求一个 MD5 值，最后我们选择了后者，具体的做法是在有列表缓存需要的 Model 里，引入一个 Concern：

```ruby
module Extensions
  module Cachable
    extend ActiveSupport::Concern

    included do
      def self.cache_key
        keys = to_a.map(&:cache_key)
        "#{table_name}/list:#{Digest::MD5.hexdigest keys.join}"
      end
    end
  end

end
```

这样，在需要对列表进行缓存的时候，我们的写法就不再是 <% cache [:todolists, @todolists.max(&:updated_at)] %>，而是这样：

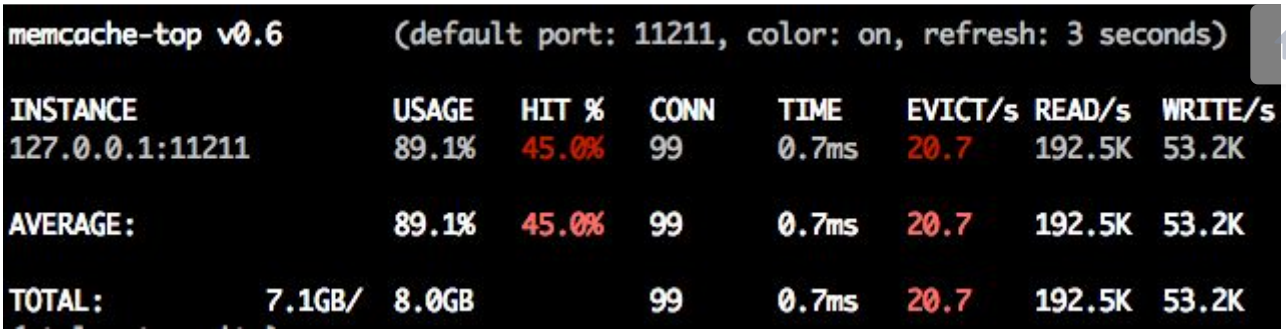
```erb
<!-- Section Todolists -->
<% cache [:todolists, @todolists.cache_key] do %>
  <%= render partial: 'todolists/todolist', collection: @todolists %>
<% end %>
```

这种办法是目前我们能想到的最佳解决方案，不知道有没有更好的处理方式。

绕过这个最大的坑以后，还剩下最后一个地方需要修改，就是我们最外层的那个套娃，我们使用的是 <% cache @project %> 来对整个项目详情页做缓存的，但是因为引入了访客锁，所以访客看到的页面，和普通成员以及管理员看到的页面，是不一样的，如果都用 @project 作为 cache_key，会导致和上面列表模式一样的问题，好在这个地方的解决方法比较简单，把缓存改成 <% cache [@project, current_user.visitor?] %> 即可，只是对于同一个项目详情页，需要存储两份缓存了。

5. 小结

以上就是我们在 Tower 里使用套娃缓存的一些经验，除了 Fragment Caching 之外，我们也没有额外再使用 Page Caching 或者 Action Caching 之类的技术，37signals 在这篇 Blog 里 统计过他们使用套娃后的缓存命中率，这个值是 67%，而 Tower 目前 8G 的 Memcache 的缓存命中率是 45%，相比之下还有差距，不过整站使用体验上，速度并不是一个显著的短板，如果能把 Fragment Cache 的细粒度继续做下去，应该会有更好的效果。



综上，套娃缓存机制还是蛮适合于小团队用来加速自己的网站（实际上 Tower 的 Hybird 模式的移动客户端也是用这种方式来做加速的），只要在模板设计的时候，尽量按照资源做好规划，后面逐步增加套娃的数量和层级，由于只涉及到模板部分的更改，总体来说是一个性价比很高的方案。

编辑于 2014-09-12