在Django中实现一个高性能未读消息计数器

04-28 00:52
3.42k

计数器(Counter)是一个非常常用的功能组件，这篇blog以未读消息数为例，介绍了在 Django中实现一个高性能计数器的基本要点。

故事的开始：.count()

假设你有一个Notification Model类，保存的主要是所有的站内通知：

```
class Notification(models.Model):
    """一个简化过的Notification类，拥有三个字段：

    - `user_id`: 消息所有人的用户ID
    - `has_readed`: 表示消息是否已读
    """

    user_id = models.IntegerField(db_index=True)
    has_readed = models.BooleanField(default=False)
```

理所当然的，刚开始你会通过这样的查询来获取某个用户的未读消息数：

```
# 获取ID为3074的用户的未读消息数
Notification.objects.filter(user_id=3074, has_readed=False).count()
```

当你的Notification表比较小的时候，这样的方式没有任何的问题，但是慢慢的，随着业务量 的扩大。消息表里面有了 上亿条数据 。很多懒惰的用户的未读消息数都到了上千条。

这时候，你就需要实现一个计数器，让这个计数器来统计每个用户的未读消息数，这样 比起之前的 `count()` ，我们只需要执行一条简单的主键查询（或者更优）就可以拿到实时的未读消息数了。

更优的方案：建立计数器

首先，让我们得建立一个新表来存储每个用户的未读消息数。

```
class UserNotificationsCount(models.Model):
    """这个Model保存着每一个用户的未读消息数目"""

    user_id = models.IntegerField(primary_key=True)
    unread_count = models.IntegerField(default=0)

    def __str__(self):
        return '<UserNotificationsCount %s: %s>' % (self.user_id, self.unread_cou
```

我们为每一个注册用户提供了一条对应的 `UserNotificationsCount` 记录来保存他的未读消息数。 每次获取他的未读消息数的时候，只需要 `UserNotificationsCount.objects.get(pk=user_id).unread_count` 就可以了。

接下来，问题的重点来了，我们如何知道什么时候应该更新我们的计数器？Django在这方面提供了什么捷径吗？

挑战：实时更新你的计数器

为了让我们的计数器正常的工作，我们必须实时的更新它，这包括：

1. 当有新的未读消息过来的时候，为计数器 **+1**
2. 当消息被异常删除时，如果关联的消息为未读，为计数器 **-1**
3. 当阅读完一个新消息的时候，为计数器 **-1**

让我们一个一个来解决这些情况。

在抛出解决方案之前，我们需要先介绍Django中的一个功能： Signals ，Signals是django提供的一个事件通知机制，它可以让你在监听某些自定义或者 预设的事件，当这些事件发生的时候，调用实现定义好的方法。

比如 `django.db.models.signals.pre_save & django.db.models.signals.post_save` 表示的是某个Model调用save方法之前和之后会触发的事件，它和Database提供的触发器在功能上有一点相似。

关于Signals更多的介绍可以参考官方文档，下面让我们来看看Signals能给我们的计数器带来什么好处。

1. 当有新的消息过来的时候，为计数器 +1

这个情况应该是最好处理的，使用Django的Signals，只需要短短几行代码，我们便可以实现这种情况下的计数器更新：

```
from django.db.models.signals import post_save, post_delete

def incr_notifications_counter(sender, instance, created, **kwargs):
    # 只有当这个instance是新创建，而且has_readed是默认的false才更新
    if not (created and not instance.has_readed):
        return

    # 调用 update_unread_count 方法来更新计数器 +1
    NotificationController(instance.user_id).update_unread_count(1)

# 监听Notification Model的post_save信号
post_save.connect(incr_notifications_counter, sender=Notification)
```

这样，每当你使用 `Notification.create` 或者 `.save()` 之类的方法创建新通知 时，我们的NotificationController 便会得到通知，为计数器 +1。

但是请注意，因为我们的计数器是基于Django的signals，如果你的代码里面有地方 在使用原始sql，没有通过Django ORM方法来添加新通知的话，我们的计数器是不会得到 通知的，所以，最好规范所有的新通知建立方式，比如使用同一个API。

2. 当消息被异常删除时，如果关联的消息为未读，为计数器 -1

有了第一个的经验，这种情况处理起来也比较简单，只需要监控Notification的post_delete 信号就可以了，下面是一段实例代码：

```
def decr_notifications_counter(sender, instance, **kwargs):
    # 当删除的消息还没有被读过时，计数器 -1
    if not instance.has_readed:
        NotificationController(instance.user_id).update_unread_count(-1)

post_delete.connect(decr_notifications_counter, sender=Notification)
```

至此，Notification的删除事件也能正常的更新我们的计数器了。

3. 当阅读一个新消息的时候，为计数器 -1

接下来，当用户阅读某条未读消息的时候，我们也需要更新我们的未读消息计数器。 你可能会说，这有什么难的？我只要在我的阅读消息的方法里面，手动更新我的计数器不就好了？

比如这样：

```
class NotificationController(object):

    ... ...

    def mark_as_readed(self, notification_id):
        notification = Notification.objects.get(pk=notification_id)
        # 没有必要重复标记一个已经读过的通知
        if notication.has_readed:
            return

        notification.has_readed = True
        notification.save()
        # 在这里更新我们的计数器，嗯，我感觉好极了
        self.update_unread_count(-1)
```

通过一些简单的测试，你可以会觉得你的计数器工作的非常好，但是，这样的实现方式有一个 非常致命的问题， 这个方式没有办法正常处理并发的请求 。

打一个比方，你拥有一个id为100的未读消息对象，这个时候同时有了两个请求过来，都要标记这个通知为已读：

```
# 因为两个并发的请求，假设这两个方法几乎同时被调用
NotificationController(user_id).mark_as_readed(100)
NotificationController(user_id).mark_as_readed(100)
```

显而易见的，这两次方法都会成功的标记这条通知为已读，因为在并发的情况下，`if notification.has_readed` 这样的检查无法正常工作，所以我们的计数器将会被错误的 -1 两次，但其实我们只读了一条请求。

那么，这样的问题应该怎么解决呢？

基本上，解决并发请求产生的数据冲突只有一个办法： 加锁，介绍两种比较简单的解决方案：

使用 `select for update` 数据库查询

`select ... for update` 是数据库层面上专门用来解决并发取数据后再修改的场景的，主流的关系数据库 比如mysql、postgresql都支持这个功能， 新版的Django ORM甚至直接提供了这个功能的shortcut 。 关于它的更多介绍，你可以搜索你使用的数据库的介绍文档。

使用 `select for update` 后，我们的代码可能会变成这样：

```
from django.db import transaction

class NotificationController(object):

    ... ...

    def mark_as_readed(self, notification_id):
        # 手动让select for update和update语句发生在一个完整的事务里面
        with transaction.commit_on_success():
            # 使用select_for_update来保证并发请求同时只有一个请求在处理，其他的请求
            # 等待锁释放
            notification = Notification.objects.select_for_update().get(pk=notificat
            # 没有必要重复标记一个已经读过的通知
            if notication.has_readed:
                return

            notification.has_readed = True
            notification.save()
            # 在这里更新我们的计数器，嗯，我感觉好极了
            self.update_unread_count(-1)
```

除了使用``select for update``这样的功能，还有一个比较简单的办法来解决这个问题。

使用**update**来实现原子性修改

其实，更简单的办法，只要把我们的数据库改成单条的update就可以解决并发情况下的问题了：

```
def mark_as_readed(self, notification_id):
        affected_rows = Notification.objects.filter(pk=notification_id, has_readed=F
```

```
                                        .update(has_readed=True)
        # affected_rows将会返回update语句修改的条目数
        self.update_unread_count(affected_rows)
```

这样，并发的标记已读操作也可以正确的影响到我们的计数器了。

高性能？

我们在之前介绍了如何实现一个能够正确更新的未读消息计数器，我们可能会直接使用UPDATE 语句来修改我们的计数器，就像这样：

```
from django.db.models import F

def update_unread_count(self, count)
    # 使用Update语句来更新我们的计数器
    UserNotificationsCount.objects.filter(pk=self.user_id)\
                                  .update(unread_count=F('unread_count') + count)
```

但是在生产环境中，这样的处理方式很有可能造成严重的性能问题，因为如果我们的计数器在频繁更新的话，海量的Update会给数据库造成不小的压力。所以为了实现一个高性能的计数器，我们 需要把改动暂存起来，然后批量写入到数据库。

使用 redis 的 sorted set ，我们可以非常轻松的做到这一点。

使用**sorted set**来缓存计数器改动

redis是一个非常好用的内存数据库，其中的sorted set是它提供的一种数据类型：有序集合， 使用它，我们可以非常简单的缓存所有的计数器改动，然后批量回写到数据库。

```
RK_NOTIFICATIONS_COUNTER = 'ss_pending_counter_changes'

def update_unread_count(self, count):
    """修改过的update_unread_count方法"""
    redisdb.zincrby(RK_NOTIFICATIONS_COUNTER, str(self.user_id), count)

# 同时我们也需要修改获取用户未读消息数方法，使其获取redis中那些没有被回写
# 到数据库的缓冲区数据。在这里代码就省略了
```

通过以上的代码，我们把计数器的更新缓冲在了redis里面，我们还需要一个脚本来把这个缓冲区 里面的数据定时回写到数据库中。

通过自定义django的command，我们可以非常轻松的做到这一点：

```
# File: management/commands/notification_update_counter.py

# -*- coding: utf-8 -*-
from django.core.management.base import BaseCommand
```

```
from django.db.models import F

# Fix import prob
from notification.models import UserNotificationsCount
from notification.utils import RK_NOTIFICATIONS_COUNTER
from base_redis import redisdb

import logging
logger = logging.getLogger('stdout')


class Command(BaseCommand):
    help = 'Update UserNotificationsCounter objects, Write changes from redis t

    def handle(self, *args, **options):
        # 首先，通过 zrange 命令来获取缓冲区所有修改过的用户ID
        for user_id in redisdb.zrange(RK_NOTIFICATIONS_COUNTER, 0, -1):
            # 这里值得注意，为了保证操作的原子性，我们使用了redisdb的pipeline
            pipe = redisdb.pipeline()
            pipe.zscore(RK_NOTIFICATIONS_COUNTER, user_id)
            pipe.zrem(RK_NOTIFICATIONS_COUNTER, user_id)
            count, _ = pipe.execute()
            count = int(count)
            if not count:
                continue

            logger.info('Updating unread count user %s: count %s' % (user_id, cou
            UserNotificationsCount.objects.filter(pk=obj.pk)\
                                          .update(unread_count=F('unread_count') +
```

之后，通过 `python manage.py notification_update_counter` 这样的命令就可以把缓冲区 里面的改动批量回写到数据库了。我们还可以把这个命令配置到crontab中来定义执行。

总结

文章到了这里，一个简单的“高性能”未读消息计数器算是实现完了。说了这么多，其实主要的知识点就是这么些：

- 使用Django的signals来获取Model的新建/删除操作更新
- 使用数据库的select for update来正确处理并发的数据库操作
- 使用redis的sorted set来缓存计数器的修改操作

希望能对您有所帮助。 :)