Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего профессионального образования «Уральский федеральный

университет имени первого президента России Б. Н. Ельцина»

Институт фундаментального образования

Кафедра интеллектуальных информационных технологий

**Сессии**

**Пояснительная записка**

Руководитель: Д.М. Свинцов

Выполнили: Р.М. Казаков

В.М. Глушкова

Екатеринбург 2017

Оглавление

[Введение 3](#_Toc477713945)

[Пример добавления CSRF токена 4](#_Toc477713946)

[Заменить пример в разделе "Всплывающие сообщения" 4](#_Toc477713947)

[Резюме 4](#_Toc477713948)

[Добавить пример использования сессий пирамиды в других фреймворках 5](#_Toc477713949)

[Сессии на Beaker 8](#_Toc477713950)

[dogpile.cache 14](#_Toc477713951)

[Список литературы 31](#_Toc477713952)

# Введение

Python — универсальный язык программирования, применимый, в том числе и в вебе. С технической точки зрения web-приложение на Python — полноценное приложение, загруженное в память, обладающее своим внутренним состоянием, сохраняемым от запроса к запросу.

Сессии предоставляют место для сохранения данных в веб-приложениях, система сеанса упрощает сеанс реализации, предоставляя WSGI промежуточное программное обеспечение, которое обрабатывает их.

# Пример добавления CSRF токена

<div style="display:none">

<input type="hidden" name="csrfmiddlewaretoken" value="$csrf\_token"/>

</div>

# Заменить пример в разделе "Всплывающие сообщения"

Для добавления сообщения в очередь флэш сообщений, используется метод flash():

request.session.flash('mymessage')

Метод flash () добавляет сообщение в очередь, если это необходимо.

flash(message, queue='', allow\_duplicate=True)

Аргумент сообщение обязательно требуется. Он представляет собой сообщение, которое вы хотите отобразить пользователю. Это, как правило, строка, но сообщение, которое вы предоставляете, не изменяется каким-либо образом.

Аргумент очереди позволяет выбрать очередь, в которую добавить сообщение, которое вы предоставляете. Это может быть использовано, чтобы толкать различные виды сообщений, в флэш-памяти для последующего отображения в разных местах на странице. Вы можете передать любое имя для вашей очереди, но оно должно быть строкой. Каждая очередь не зависима, и может быть очищен с помощью метода pop\_flash () или с помощью peek\_flash () отдельно. По умолчанию очередь — это пустая строка. Пустая строка представляет очередь мгновенное сообщение по умолчанию.

request.session.flash(msg, 'myappsqueue')

request.session.pop\_flash()

request.session.peek\_flash()

# Резюме

|  |
| --- |
| Скрипт для обновления резюме |
|  | import requests |
|  |  |
|  | def fun(): |
|  | access\_token = '\*\*\*\*\*' #your acces token |
|  | url = 'https://api.hh.ru/resumes/\*\*\*\*\*/publish' #your resume id |
|  | response = requests.post( |
|  | url , |
|  | headers={ |
|  | 'User-Agent': 'Test/1.0 (\*\*\*\*\*@gmail.com)', #your @mail |
|  | 'Content-Type': 'application/json', |
|  | 'Authorization': 'Bearer %s' % access\_token |
|  | } |
|  | ) |
|  | print(response.status\_code) |
|  | x = fun() |
|  | print('finish') |

# Добавить пример использования сессий пирамиды в других фреймворках

Сессии в django

Django полностью поддерживает сессии для анонимных пользователей, позволяет сохранять и получать данные для каждой посетителя сайта. Механизм сессии сохраняет данные на сервере и самостоятельно управляет сессионными куками. Куки содержат ID сессии, а не сами данные.

Активация

Сессии реализованы с помощью отдельного модуля и моделей Django. Для активации сессий выполните следующие шаги:

Измените параметр MIDDLEWARE\_CLASSES, добавив в него значениеdjango.contrib.sessions.middleware.SessionMiddleware.

Внесите в параметр INSTALLED\_APPS значение django.contrib.sessions. Запустите manage.py syncdb после этого.

Стандартный шаблон приложения, созданный с помощью startproject уже имеет все необходимые настройки и если вы ничего не удаляли, то ничего добавлять не понадобится для работы сессий.

Если вам не требуется использовать сессии, вы можете удалить строку SessionMiddleware из параметраMIDDLEWARE\_CLASSES и строку django.contrib.sessions из параметра INSTALLED\_APPS. Это позволит вам освободить крайне незначительную часть ресурсов, но из маленьких частей складываются большие.

Использование в функциях представления

После активации SessionMiddleware каждый объект HttpRequest — первый аргумент любой функции представления Django — будет иметь атрибут session по функциональности аналогичный словарю. Пример:

# Установим значение ключа в сессии:

request.session["fav\_color"] = "blue"

# Получим значение ключа из сессии -- можно это сделать в

# другом представлении или через несколько запросов

# (или и то и другое):

fav\_color = request.session["fav\_color"]

# Чистим значение ключа в сессии:

del request.session["fav\_color"]

# Проверка наличия ключа в сессии:

if "fav\_color" in request.session:

...

Вы также можете использовать другие методы, такие как keys() и items() объекта request.session.

Есть ряд простых правил для эффективного использования сессий в Django:

Используйте для объекта request.session обычные строки Python в качестве ключей словаря. Это правило служит больше для удобства, чем для ускорения работы, но оно стоит того, чтобы ему следовать.

Словарные ключи сессии, которые начинаются с символа «подчёркивание», зарезервированы для внутреннего использования Django. На практике среда использует небольшой набор зарезервированных переменных в сессии, но пока вы не знаете об их назначении, не стоит их трогать.

Не заменяйте request.session новым объектом и не работайте с его атрибутами. Используйте его как обычный словарь Python.

Рассмотрим несколько примеров. Нижеприведённая простейшая функция представления устанавливает переменной has\_commented значение True после того, как пользователь отправляет комментарий. Это простой (но небезопасный) способ предотвращения повторной отправки пользовательского комментария:

def post\_comment(request, new\_comment):

if request.session.get('has\_commented', False):

return HttpResponse("Вы уже отправили комментарий")

c = comments.Comment(comment=new\_comment)

c.save()

request.session['has\_commented'] = True

return HttpResponse('Спасибо за ваш комментарий!')

Это простейшее представление аутентифицирует пользователя на сайте:

def login(request):

try:

m = Member.objects.get(username\_\_exact=request.POST['username'])

if m.password == request.POST['password']:

request.session['member\_id'] = m.id

return HttpResponse("Вы авторизованы.")

except Member.DoesNotExist:

return HttpResponse("Ваши логин и пароль не соответствуют.")

Это представление отключает ранее авторизованного пользователя:

def logout(request):

try:

del request.session['member\_id']

except KeyError:

pass

return HttpResponse("Вы вышли.")

# Сессии на Beaker

Все куки подписаны с подписью HMAC для предотвращения несанкционированного доступа клиента.

**Lazy-Loading**

Только тогда, когда объект сеанса доступа к ним будет загружен из файловой системы, предотвращая удары производительности на страницах, которые не используют сеанс.

**Применение**

Объект сеанса обеспечивается Beaker “SessionMiddleware” реализует dict-style интерфейс с несколькими дополнительными методами объекта. После того, как “SessionMiddleware” на месте, объект сеанса будет доступен как “beaker.sessionв” в WSGI среде.

Получение данных из сессии:

myvar = session['somekey']

Тестирование для значения:

logged\_in = 'user\_id' in session

Добавление данных в сессии:

session['name'] = 'Fred Smith'

Полный пример с использованием основного приложения WSGI с сессиями:

from beaker.middleware import SessionMiddleware

def simple\_app(environ, start\_response):

# Get the session object from the environ

session = environ['beaker.session']

# Check to see if a value is in the session

user = 'logged\_in' in session

# Set some other session variable

session['user\_id'] = 10

start\_response('200 OK', [('Content-type', 'text/plain')])

return ['User is logged in: %s' % user]

# Configure the SessionMiddleware

session\_opts = {

'session.type': 'file',

'session.cookie\_expires': True,

}

wsgi\_app = SessionMiddleware(simple\_app, session\_opts)

**Атрибуты сессии / Ключи**

Сессии имеют несколько специальных атрибутов, которые могут быть использованы в случае необходимости с помощью приложения.

* ID - уникальный 40-символьный SHA-сгенерированный идентификатор сеанса
* last\_accessed - В последний раз сессия была открыта до текущего доступа, если save\_accessed\_time верно; в последний раз он был изменен, если ложно; сеанс был только что сделал.

Есть несколько специальных ключей сессии, заполняются так:

* \_accessed\_time - Сеанс был загружен, если save\_accessed\_time верно; она была написана в прошлом, если ложь
* \_creation\_time - Когда сессия была создана.

**Сохранение**

Сессии могут быть сохранены с помощью save()метода объекта сессии:

session.save()

Этот флаг сессии будет сохранен, и будет храниться на выбранном месте в конце запроса.

Если необходимо немедленно сохранить сессию к фоновым, то то должен использоваться метод persist():

session.persist()

Обычно это не так, однако, как правило, сессия не должна быть сохранена, должно произойти что-то катастрофическое во время запроса.

Порядок имеет значение: при использовании промежуточного Beaker, необходимо вызвать сохранить, прежде чем заголовки отправляются клиенту. С Beaker’s промежуточного наблюдения, когда функция start\_response вызывается, чтобы знать, что он должен добавить его заголовок файла cookie, сессии должны быть сохранены, прежде чем он вызывается.

Объекты реагирования в рамках популярных (WebOb, WERKZEUG и т.д.) start\_response вызова немедленно, так что если вы используете один из этих объектов для обработки вашего ответа, вы должны вызвать .save () до того, как объект Response вызывается:

# this would apply to WebOb and possibly others too

from werkzeug.wrappers import Response

# this will work

def sessions\_work(environ, start\_response):

environ['beaker.session']['count'] += 1

resp = Response('hello')

environ['beaker.session'].save()

return resp(environ, start\_response)

# this will not work

def sessions\_broken(environ, start\_response):

environ['beaker.session']['count'] += 1

resp = Response('hello')

retval = resp(environ, start\_response)

environ['beaker.session'].save()

return retval

**Автосохранение**

Сохранение может быть сделано автоматически, установив auto параметр конфигурации для сеансов. Когда установлено, вызывать save()метод больше не требуется, и сеанс будет сохранен автоматически в любое время при обращении к нему во время запроса.

**Удаление**

Вызов delete()метода удаляет сессию из серверного хранения и посылает на истечение куки браузера с просьбой, на его очистку:

session.delete()

Это должно быть использовано в конце запроса, когда сеанс будет удален и не будет в дальнейшем использоваться в запросе.

Если сессия должна быть признана недействительной, и новый сеанс создается и используется во время запроса, должен использоваться метод invalidate():

session.invalidate()

**Снятие с истекшим сроком действия**

Beaker вовсе не автоматически удалять старые или с истекшим сроком действия куки на любом из его обратных концах. Эта задача остается на усмотрение разработчика на основе того, как используются сеансы, а также на том, что бэк-энда.

База данных бэкэнда записи последнего времени доступа как столбца в базе данных так, сценарий может быть запущен для удаления сеанса строк в базе данных, которые не были использованы в течение длительного времени.

При использовании сессий на основе файлов, скрипт может работать, чтобы удалить файлы, которые не были затронуты в течение длительного времени, например (в директории данных дня сессии):

find . -mtime +3 -exec rm {} \;

**Домен cookie и Путь**

В дополнение к установке домена cookie по умолчанию с установкой домена cookie , домен куки и путь может быть установлен динамически для сеанса со свойствами домена и пути.

Эти настройки будут сохраняться до тех пор, пока не существует куки, или до изменения.

Пример:

# Setting the session's cookie domain and path

session.domain = '.domain.com'

session.path = '/admin'

**Cookie-Based**

Сессия может храниться исключительно на стороне клиента с помощью куки на основе сеансов. Эта опция может быть включена путем установки типа сеанса cookie.

Использование куки на основе сеанса несет ограничение, насколько большой куки может быть (как правило, 4096 байт). Исключение будет выдано если сессии слишком большие, чтобы поместиться в куки, так что использование куки на основе сеанса должно быть сделано тщательно и только небольшие биты данных должны быть сохранены в них (пользователи вводят логин и имя, статус администратора и т.д.).

Большие куки могут замедлить загрузку страниц, поскольку они увеличивают задержку на каждый запрос страницы, так как куки отправляются для каждого запроса в соответствии с доменом. Статическое содержание таких как изображения и Javascript должно быть подано от домена, что куки не является допустимым, чтобы предотвратить это.

Сеансы Cookie на основе легко масштабируются в кластерной среде, поскольку нет никакой необходимости в общей системе хранения, когда различные серверы обработки той же сессии.

**Шифрование**

В том случае, если куки на основе сеансов также должны быть зашифрованы, чтобы запретить пользователю возможность декодировать данные (в дополнение к не в состоянии подделать с ним), Beaker может использовать 256-битное AES шифрование для защиты содержимого куки.

В зависимости от реализации Python, Beaker может потребоваться дополнительная библиотека для шифрования AES.

На CPython (регулярный Python), требуется одна из следующих библиотек:

* [Python-NSS](https://pypi.python.org/pypi/python-nss/) библиотека
* [Pycryptopp](https://pypi.python.org/pypi/pycryptopp/) библиотека
* [Криптография](https://pypi.python.org/pypi/cryptography/) библиотека
* [PyCrypto](https://pypi.python.org/pypi/pycrypto/) библиотека

На Jython, никаких дополнительных пакетов не требуется, но, по крайней мере на Sun JRE, размер ключа шифрования по умолчанию ограничен до 128 бит, что приводит к генерируемым сеансам несовместимым с теми, генерируются в CPython, и наоборот. Чтобы преодолеть это ограничение, вам необходимо установить файлы юрисдикции политики безграничную силу от Sun:

* [Policy files for Java 5](https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jce_policy-1.5.0-oth-JPR@CDS-CDS_Developer)
* [Policy files for Java 6](https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jce_policy-6-oth-JPR@CDS-CDS_Developer)

# dogpile.cache

**Обзор**

С помощью системы кэширования в памяти, dogpile.cache предоставляет интерфейс для конкретного Python API, ориентированного на эту систему.

Конфигурация dogpile.cache состоит из следующих компонентов:

* *Область*, которая является экземпляром **[CacheRegion](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion" \o "dogpile.cache.region.CacheRegion)**, и определяет информацию о конфигурации для конкретного бэкэнда. **[CacheRegion](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion" \o "dogpile.cache.region.CacheRegion)** Можно считать "front end" используемым приложениями.
* *Бэкенд*, который является экземпляром **[CacheBackend](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.api.CacheBackend" \o "dogpile.cache.api.CacheBackend)**, описания того, как значения сохраняются и извлекаются из внутреннего интерфейса. Этот интерфейс определяет только get(), set()и delete(). Фактический вид CacheBackend используется для конкретной CacheRegion определяется в основе API Python используется, чтобы поговорить с кэшем, таким как Pylibmc. CacheBackend Инстанциируется за кулисами, а не непосредственно доступ к ним при нормальных обстоятельствах.
* Функции генерации значений. Эти определяемые пользователем функции, которые генерируют новые значени, которые будут помещены в кэш. В то время как dogpile.cache предлагает обычный "набор" подход размещения данных в кэш, обычный режим использования является лишь инструктированием его "получить" значения, передавая ей *функцию создания*, которая будет использоваться для генерации нового значения, тогда и только тогда, когда он требуется. Это "получить или создать" шаблоном является весь ключ к системе "Dogpile", который координирует одну операцию создания стоимости, среди многих операций одновременно получая для конкретного ключа, что исключает проблемы с истекшим сроком действия значения избыточного регенеририрования многих работников одновременно.

**Rudimentary Usage**

dogpile.cache включает в себя базовую программу Pylibmc. Базовая конфигурация выглядит следующим образом:

from dogpile.cache import make\_region

region = make\_region().configure(

'dogpile.cache.pylibmc',

expiration\_time = 3600,

arguments = {

'url': ["127.0.0.1"],

}

)

@region.cache\_on\_arguments()

def load\_user\_info(user\_id):

return some\_database.lookup\_user\_by\_id(user\_id)

Выше, мы создаем CacheRegion c помощью make\_region()функции, а затем применяем конфигурацию бэкэнд через CacheRegion.configure()метод, который возвращает регион. Вызванный бэкэнд является единственным аргументом, этого требует CacheRegion.configure() сам по себе, и в этом случае dogpile.cache.pylibmc. Тем не менее, в данном конкретном случае, pylibmc бэкенд также требует, чтобы URL - адрес на Memcached сервера передавался в arguments словаре.

Конфигурация разделяется на две секции. После завершения строительства через make\_region(), то CacheRegion объект будет доступен, как правило, во время импорта модуля, для использования в функции декорирования. Дополнительные сведения о конфигурации, передаваемые CacheRegion.configure()обычно загружаются из конфигурационного файла и поэтому не обязательно доступны до выполнения, следовательно, конфигурация - двухэтапный процесс.

Ключевые аргументы передаются CacheRegion.configure()включают EXPIRATION\_TIME , который является временем окончания срока передачи в замок Dogpile, и аргументы , которые являются аргументами, используемые непосредственно бэкэндом - в данном случае мы используем аргументы, которые передаются непосредственно к модулю pylibmc.

**Region Configuration**

make\_region()Функция в настоящее время вызывает CacheRegion конструктор напрямую.

Класс dogpile.cache.region.CacheRegion( имя = None , function\_key\_generator = <функция function\_key\_generator> , function\_multi\_key\_generator = <функция function\_multi\_key\_generator> , key\_mangler = None , async\_creation\_runner = None )

Параметры:

|  |  |  |  |
| --- | --- | --- | --- |
|  | * **Имя** - Необязательно, имя строки для региона. Не используется внутри страны, но могут быть доступны через .name параметр, полезно для настройки региона из файла конфигурации. * **function\_key\_generator** - Необязательный. Функция, которая будет производить "ключ кэша" задана функцией создания данных и аргументов, при использовании **[CacheRegion.cache\_on\_arguments()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion.cache_on_arguments" \o "dogpile.cache.region.CacheRegion.cache_on_arguments)** метода. Структура этой функции имеет два уровня: при заданной функции создания данных, возвращают новую функцию, которая генерирует ключ на основе заданных аргументов. Такие как:   def my\_key\_generator(namespace, fn, \*\*kw):  fname = fn.\_\_name\_\_  def generate\_key(\*arg):  return namespace + "\_" + fname + "\_".join(str(s) for s in arg)  return generate\_key  region = make\_region(  function\_key\_generator = my\_key\_generator  ).configure(  "dogpile.cache.dbm",  expiration\_time=300,  arguments={ "filename":"file.dbm"  }  )  Namespace Является то, что передается CacheRegion.cache\_on\_arguments(). Это не консультации вне этой функции, так что на самом деле может быть любой формы. Например, он может быть передан как кортеж, используемый для указания аргументов от \*\* кВт:  def my\_key\_generator(namespace, fn):  def generate\_key(\*arg, \*\*kw):  return ":".join(  [kw[k] for k in namespace] +  [str(x) for x in arg]  )  return generate\_key  Там, где декоратор может быть использован в качестве:  @my\_region.cache\_on\_arguments(namespace=('x', 'y'))  def my\_function(a, b, \*\*kw):  return my\_data()   * **function\_multi\_key\_generator  -** Необязательный. Аналогично function\_key\_generator параметру**,**  но он используется в CacheRegion.cache\_multi\_on\_arguments(). Сформированная функция должна возвращать список ключей. Например:   def my\_multi\_key\_generator(namespace, fn, \*\*kw):  namespace = fn.\_\_name\_\_ + (namespace or '')  def generate\_keys(\*args):  return [namespace + ':' + str(a) for a in args]  return generate\_keys   * **key\_mangler** - функция, которая будет использоваться на всех входящих ключах, прежде чем перейти к бэкэндам. По умолчанию будет использоваться None , в этом случае ключевая функция коверкания рекомендована бэкэндом кэша. Типичный Mangler является SHA1 Mangler найден [**sha1\_mangle\_key()**](http://dogpilecache.readthedocs.io/en/latest/api.html#dogpile.cache.util.sha1_mangle_key) на котором ключи принуждает в SHA1 хэш, таким образом, чтобы длина строки была фиксирована. Чтобы отключить все ключевые коверкания, установите False. Другим типичным Mangler является встроенный в Python функция str, которая может быть использована для преобразования не строковых или Unicode ключи для байтовых строк, которая необходима при использовании бэкэнда, таких как bsddb или дБмВт под Python 2.x в сочетании с клавишами Unicode. * async\_creation\_runner - Отзывной, что, если он указан, будет передан и названный dogpile.lock, когда четкое значение присутствует в кэше. Он будет передан мьютекс и отвечать, отпуская эту взаимную блокировку, когда закончите. Это может быть использовано, чтобы отложить вычисление дорогих функций на более поздние моменты другим путем, например, фоновый поток, затянувшийся очереди, или менеджер задач, как сельдерей.   Для конкретного примера, используя async\_creation\_runner, новые значения могут быть созданы в фоновом потоке, так как:  import threading  def async\_creation\_runner(cache, somekey, creator, mutex):  ''' Used by dogpile.core:Lock when appropriate '''  def runner():  try:  value = creator()  cache.set(somekey, value)  finally:  mutex.release()  thread = threading.Thread(target=runner)  thread.start()  region = make\_region(  async\_creation\_runner=async\_creation\_runner,  ).configure(  'dogpile.cache.memcached',  expiration\_time=5,  arguments={  'url': '127.0.0.1:11211',  'distributed\_lock': True,  }  )  Если у вас есть один CacheRegion, то CacheRegion.cache\_on\_arguments()способ может быть использован для украшения функции, но сама кэш не может быть использован до тех пор , CacheRegion.configure() называется. Интерфейс для этого способа заключается в следующем:CacheRegion.configure( Бэкенд , EXPIRATION\_TIME = None , аргументы = None , \_config\_argument\_dict = None , \_config\_prefix = None , упаковка = None , replace\_existing\_backend = False , region\_invalidator = None )  Настроить CacheRegion.  CacheRegionСам возвращается.  Параметры:   * **бэкенд** - Обязательно. Это имя из **[CacheBackend](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.api.CacheBackend" \o "dogpile.cache.api.CacheBackend)** , и решается путем загрузки класса из dogpile.cacheEntryPoint. * **EXPIRATION\_TIME** - Необязательный. Время окончания срока передается в систему Dogpile. Может быть передано как целое число секунд, или в качестве datetime.timedelta значения.   [**CacheRegion.get\_or\_create()**](http://dogpilecache.readthedocs.io/en/latest/api.html#dogpile.cache.region.CacheRegion.get_or_create) Метод, а также **[CacheRegion.cache\_on\_arguments()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion.cache_on_arguments" \o "dogpile.cache.region.CacheRegion.cache_on_arguments)** декоратор (хотя примечание: **не** [**CacheRegion.get()**](http://dogpilecache.readthedocs.io/en/latest/api.html#dogpile.cache.region.CacheRegion.get)метод) призыва функции создания значения после периода времени прошедшего с момента последнего поколения.   * **Аргументы** - необязательно. Структура здесь передается непосредственно к конструктору **[CacheBackend](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.api.CacheBackend" \o "dogpile.cache.api.CacheBackend)** в использовании, хотя обычно представляет собой словарь. * **Завернуть** - Необязательный. Список **[ProxyBackend](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.proxy.ProxyBackend" \o "dogpile.cache.proxy.ProxyBackend)** классов и / или экземпляров, каждый из которых будет применяться в цепи, чтобы, в конечном счете, скрыть оригинальный бэкенд, так что пользовательские функциональные дополнения могут быть применены. * **replace\_existing\_backend**- если это правда, существующий бэкэнды будет заменен. Без этого флага, исключение возникает, если бэкенд уже настроен. * **region\_invalidator** - Необязательный. Override стратегия недействительности по умолчанию с пользовательской реализацией **[RegionInvalidationStrategy](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.RegionInvalidationStrategy" \o "dogpile.cache.region.RegionInvalidationStrategy)**.   CacheRegion Также можно настроить из словаря, используя CacheRegion.configure\_from\_config() метод:  CacheRegion.configure\_from\_config( Config\_dict , префикс )  Настройка из словаря конфигурации и префикса.  local\_region = make\_region()  memcached\_region = make\_region()  # regions are ready to use for function  # decorators, but not yet for actual caching  # later, when config is available  myconfig = {  "cache.local.backend":"dogpile.cache.dbm",  "cache.local.arguments.filename":"/path/to/dbmfile.dbm",  "cache.memcached.backend":"dogpile.cache.pylibmc",  "cache.memcached.arguments.url":"127.0.0.1, 10.0.0.1",  }  local\_region.configure\_from\_config(myconfig, "cache.local.")  memcached\_region.configure\_from\_config(myconfig,  "cache.memcached.")  **Using a Region**  CacheRegion Объект является передним концом интерфейса к кэшу. Включает в себя следующие методы:  CacheRegion.get( Ключ , EXPIRATION\_TIME = None , ignore\_expiration = False )  Возвращает значение из кэша, основываясь на данном ключе.  Если значение нет, метод возвращает маркер NO\_VALUE. NO\_VALUE оценивается как ложное, но отдельно от None различают закэшированное значение None.  По умолчанию, заданного времени окончания срока действия CacheRegion, или в качестве альтернативы истечения времени , поставляемого expiration\_time аргумента, проверяется на время создания считываемого значения по отношению к текущему времени (по данным time.time()). Если старый, кэшированное значение игнорируется , и NO\_VALUE маркер возвращается. Передача флага ignore\_expiration=True обходит проверку времени истечения.  CacheRegion.get() теперь проверяет время создания Значение против времени истечения срока действия, а не возвращать значение безоговорочно.  Метод также интерпретирует кэшированное значение с точки зрения нынешнего "недействительности" время как установлено invalidate()методом. Если значение присутствует, но время его создания старше, чем текущее время недействительности, то NO\_VALUE маркер возвращается. Передача флага ignore\_expiration=True обходит проверку времени недействительности.  Поддержка CacheRegion.invalidate() метода.  Параметры:   * **Ключ** - Ключ необходимо получить. В то время как это актуально для ключа, чтобы быть строкой, то она в конечном счете передается непосредственно вниз к бэкэндам кэша, перед тем , необязательно обработаннаяс помощью функции key\_mangler, так что может быть любого типа , признанного внутреннего интерфейса или с помощью функции key\_mangler, если он присутствует. * **EXPIRATION\_TIME** - Дополнительное значение времени истечения, которое вытеснит, что CacheRegion настроен на себя. * **ignore\_expiration** - если True значение будет возвращено из кэша, если присутствует, независимо от того, сконфигурированно время истечения срока действия или нет, то будет вызван invalidate().   CacheRegion.get\_or\_create(key, creator, expiration\_time=None, should\_cache\_fn=None)  Возвращает кэшированное значение, основанное на данном ключе.  Если значение не существует или считается, истек срок действия на основании его времени создания, данная функция создания может или не может быть использована для воссоздания значения и сохраняются вновь сформированное значение в кэше.  Является ли функция используемой , зависит от того, если замок Dogpile может быть приобретен или нет. Если он не может, это значит, другой поток или процесс уже запустил функцию создания для этого ключа от кэша. Когда замок Dogpile не может быть получен, этот метод будет блокирован, если нет предыдущее значение не доступно, пока блокировка не будет выпущен и допускает новое значение. Если предыдущее значение доступно, это значение сразу же возвращается без блокировки.  Если invalidate()метод был вызван, и отметка времени извлеченным значением является старше , чем недействительность метки времени, значение безусловно предотвращается возвращается. Метод будет пытаться получить блокировку Dogpile , чтобы создать новое значение, или будет ждать, пока блокировка не будет снята, чтобы вернуть новое значение.  Значение безусловно регенерируется, если время создания старше последнего вызова invalidate().  Параметры:   * **Ключ** - Ключ необходимо получить. В то время как это типично для ключа, чтобы быть строкой, то она в конечном счете передается непосредственно вниз к бэкэнду кэша, перед тем, необязательно обрабатывание с помощью функции key\_mangler, так что может быть любого типа, признанного внутреннего интерфейса или с помощью функции key\_mangler, если он присутствует. * **Создатель** - функция, которая создает новое значение. * **EXPIRATION\_TIME** - время истечения срока действия, который будет overide время истечения срока действия уже настроенных на этом , **[CacheRegion](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion" \o "dogpile.cache.region.CacheRegion)** если не None. Чтобы установить срок действия не используйте значение -1. * **should\_cache\_fn** - опционально вызываемая функция, которая получит значение, возвращаемое "творца", а затем вернется истинным или ложным, указывая, если значение на самом деле должно быть в кэше или нет. Если она возвращает значение False, то значение по-прежнему возвращается, но не кэшируются. Например:   def dont\_cache\_none(value):  return value is not None  value = region.get\_or\_create("some key",  create\_value,  should\_cache\_fn=dont\_cache\_none)  Выше, функция возвращает значение create\_value (), если кэш является недействительным, однако, если возвращаемое значение нет, он не будет кэшировать.  acheRegion.set( Ключ, значение )  Поместите новое значение в кэше под данным ключом.  CacheRegion.delete( Ключ )  Удалить значение из кэша.  Эта операция является тождественной (можно назвать несколько раз, или на несуществующий ключ, безопасно)  CacheRegion.cache\_on\_arguments(namespace=None, expiration\_time=None, should\_cache\_fn=None, to\_str=<type 'str'>, function\_key\_generator=None)  Функция декоратора, которая будет кэшировать возвращаемое значение функции с помощью ключа, полученного из самой функции и ее аргумента.  Декоратор внутренне использует CacheRegion.get\_or\_create()метод доступа к кэш и условно вызывает функцию. Смотрите, что метод дополнительных поведенческих деталей.  Например:  @someregion.cache\_on\_arguments()  def generate\_something(x, y):  return somedatabase.query(x, y)  Декорированная функция, то можно назвать, как правило, где данные будут выведены из региона кэша, если не требуется новое значение:  result = generate\_something(5, 6)  Функцие также дается атрибут invalidate(), который предусматривает признания не действительными значения. Переход на invalidate() те же аргументы бы перешли к самой функции, чтобы представлять особую ценность:  generate\_something.invalidate(5, 6)  Еще один атрибут set()добавляется, чтобы обеспечить дополнительные возможности кэширования по отношению к функции. Это удобный метод для CacheRegion.set() который будет хранить данное значение непосредственно без вызова декорированного функции. Значение, которое будет кэшировать передается в качестве первого аргумента, и аргументы, которые обычно передаются в функции должны следовать:  generate\_something.set(3, 5, 6)  Приведенный выше пример эквивалентен вызову, если функции должны были получить значение, как значение , которое должно быть в кэше.generate\_something(5, 6)3  Добавлен set()метод для украшенной функции.  Аналогично set()это refresh(). Этот атрибут будет вызывать декорирование функции и заполнять новое значение в кэш с новым значением, а также возвращение этого значения:  newvalue = generate\_something.refresh(5, 6)  Добавлен refresh()метода для украшенной функции.  И, наконец, get()метод возвращает либо значение кэшируют для данного ключа или маркера , NO\_VALU Eесли нет такого ключа не существует:  value = generate\_something.get(5, 6)  Добавлен get()метод для украшенной функции.  Генерация ключа по умолчанию будет использовать имя функции, имя модуля для функции, аргументы, передаваемые, а также необязательный параметр "пространства имен" для того, чтобы сгенерировать ключ кэша.  Для данной функции one внутри модуля myapp.tools:  @region.cache\_on\_arguments(namespace="foo")  def one(a, b):  return a + b  Выше, вызов будет производить ключ кэша следующим образом : one(3, 4)  myapp.tools:one|foo|3 4  Генератор ключей будет игнорировать первоначальный аргумент self или cls, что делает декоратор подходящий (с оговорками) для использования с методами экземпляра или класса. Учитывая пример:  class MyClass(object):  @region.cache\_on\_arguments(namespace="foo")  def one(self, a, b):  return a + b  Ключ кэша выше снова будет производить тот же ключ кэша - имя пропускается. MyClass().one(3, 4)myapp.tools:one|foo|3 4self  Namespace Параметр является необязательным, и используется обычно для устранения неоднозначности двух функции одного и того же имени в том же модуле, так как может произойти при декорировании метода экземпляра или класса , как показано ниже:  class MyClass(object):  @region.cache\_on\_arguments(namespace='MC')  def somemethod(self, x, y):  ""  class MyOtherClass(object):  @region.cache\_on\_arguments(namespace='MOC')  def somemethod(self, x, y):  Выше, namespace параметр устраняет неоднозначность между somemethod на MyClassи MyOtherClass. Python механика Объявление класса иным образом препятствует декоратор от необходимости осведомленности о них MyClass и MyOtherClassимен, так как функция получает декоратора, прежде чем она становится методом экземпляра.  Генерации ключа функция может быть полностью заменена на основе каждого региона, используя function\_key\_generator, аргумент присутствует **[make\_region()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.make_region" \o "dogpile.cache.region.make_region)**и **[CacheRegion](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion" \o "dogpile.cache.region.CacheRegion)**. Если по умолчанию **[function\_key\_generator()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.util.function_key_generator" \o "dogpile.cache.util.function_key_generator)**.  Параметры:   |  |  | | --- | --- | |  | * **Пространство имен** - необязательный аргумент строка, которая будет создана как часть ключа кэша. Это может потребоваться для устранения неоднозначности функции одного и того же имени в пределах того же исходного файла, такие как те, которые связаны с классами - обратите внимание, что сам по себе декоратор не может видеть родительский класс на функции как объявляется класс. * **EXPIRATION\_TIME** - если не None, будет перекрыто нормальное время истечения.Может быть определен как вызываемый, не принимая никаких аргументов, что возвращает значение, которое будет использоваться в качестве expiration\_time. Этот вызываемый будет называться всякий раз, когда сама декорированная функция вызывается, кэширование или извлечения. Таким образом, это может быть использовано для определения *динамического* времени истечения кэшированных результатов функции. Пример случая использования включают "не кэшировать результат до конца дня, недели или период времени" и "кэше до определенной даты или времени прохода".   expiration\_time может быть передан как вызываемый в **[CacheRegion.cache\_on\_arguments()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion.cache_on_arguments" \o "dogpile.cache.region.CacheRegion.cache_on_arguments)**.   * **should\_cache\_fn** - передается **[CacheRegion.get\_or\_create()](http://dogpilecache.readthedocs.io/en/latest/api.html" \l "dogpile.cache.region.CacheRegion.get_or_create" \o "dogpile.cache.region.CacheRegion.get_or_create)**. * **to\_str** - отозваны, будет называться по каждой функции аргумента для того, чтобы преобразовать в строку. Значения по умолчанию для str(). Если функция принимает аргументы юникодные не-ASCII на Python 2.x, то unicode()Builtin может быть заменен, но обратите внимание, это произведет юникодные ключи кэша, которые могут потребовать ключ коверкая до достижения кэша. * **function\_key\_generator** - функция, которая будет производить "ключ кэша". Эта функция заменит один, настроенный на себя CacheRegion. |   **Создание Бэкэндов**  Backends расположены с использованием системы Setuptools Entrypoint. Для того чтобы сделать жизнь проще, для авторов узкоспециализированных бэкэндов, вспомогательная функция включена, которая регистрирует любую бэкенд таким же образом, как если бы она была частью существующего sys.path.  Например, чтобы создать бэкенд с именем DictionaryBackend, мы создаем подкласс CacheBackend:  from dogpile.cache.api import CacheBackend, NO\_VALUE  class DictionaryBackend(CacheBackend):  def \_\_init\_\_(self, arguments):  self.cache = {}  def get(self, key):  return self.cache.get(key, NO\_VALUE)  def set(self, key, value):  self.cache[key] = value  def delete(self, key):  self.cache.pop(key)  Затем убедитесь, что класс доступен под EntryPoint dogpile.cache. Если бы мы сделали это в setup.py файле, это было бы в setup() качестве:  entry\_points="""  [dogpile.cache]  dictionary = mypackage.mybackend:DictionaryBackend  """  В качестве альтернативы, если мы хотим зарегистрировать плагин в том же пространстве процесса , не потрудившись устанавливать что - либо, мы можем использовать register\_backend:  from dogpile.cache import register\_backend  register\_backend("dictionary", "mypackage.mybackend", "DictionaryBackend")  Наш новый бэкенд будет использоваться в таком регионе, как этот:  from dogpile.cache import make\_region  region = make\_region("myregion")  region.configure("dictionary")  data = region.set("somekey", "somevalue")  Значения, которые мы получаем для бэкэндов, здесь являются экземплярами CachedValue. Это кортеж подкласс длины два, вида:  (payload, metadata)  Там, где "полезная нагрузка" это вещь в кэше, и "метаданные" является информацией, которую мы храним в кэше - словарь, который в настоящее время имеет только "время создания" и "идентификатор версии" в качестве ключевых значений. Если базовая кэш требует сериализации, рассол или подобное может быть использовано на кортеж - "метаданные" часть всегда будет небольшой и легко сериализуемым структура Python.  **Изменение Backend Поведения**  ProxyBackend Класс декоратора при желании, легко дополнить существующее поведение бэкэнд без необходимости расширения исходного класса. Использование класса декоратора также как adventageous позволяет разделять измененное поведение между различными бэкэндами.  Доверенные добавляются к CacheRegion объекту с помощью CacheRegion.configure() метода. Только Перегруженные методы должны быть определены и реальный бэкенд может получить доступ с self.proxied объекта изнутри ProxyBackend.  Например, простой класс для регистрации всех вызовов .set()будет выглядеть следующим образом:  from dogpile.cache.proxy import ProxyBackend  import logging  log = logging.getLogger(\_\_name\_\_)  class LoggingProxy(ProxyBackend):  def set(self, key, value):  log.debug('Setting Cache Key: %s' % key)  self.proxied.set(key, value)  ProxyBackend может быть выполнен с возможностью дополнительно принимать аргументы ( до тех пор, пока ProxyBackend.\_\_init\_\_()метод вызывается должным образом, либо непосредственно, либо с помощью super(). В приведенном ниже примере RetryDeleteProxy класс принимает retry\_count параметр при инициализации. В случае исключения на удаление (), он будет повторять попытки много раз, прежде чем вернуться:  from dogpile.cache.proxy import ProxyBackend  class RetryDeleteProxy(ProxyBackend):  def \_\_init\_\_(self, retry\_count=5):  super(RetryDeleteProxy, self).\_\_init\_\_()  self.retry\_count = retry\_count  def delete(self, key):  retries = self.retry\_count  while retries > 0:  retries -= 1  try:  self.proxied.delete(key)  return  except:  pass  wrap Параметр CacheRegion.configure()принимает список, который может содержать любую комбинацию из конкретизированных прокси - объектов, а также uninstantiated прокси - классов. Выставление два примера выше, вместе будет выглядеть следующим образом:  from dogpile.cache import make\_region  retry\_proxy = RetryDeleteProxy(5)  region = make\_region().configure(  'dogpile.cache.pylibmc',  expiration\_time = 3600,  arguments = {  'url':["127.0.0.1"],  },  wrap = [ LoggingProxy, retry\_proxy ]  )  В приведенном выше примере, LoggingProxyобъект будет instantated сама CacheRegionи применяется для обертывания запросов от имени retry\_proxy экземпляра; что прокси - сервер, в свою очередь оборачивает запросы от имени оригинального dogpile.cache.pylibmc бэкэндом. |

# Список литературы

1. https://habrahabr.ru/post/243961/

2. http://dogpilecache.readthedocs.io/en/latest/usage.html

3. http://beaker.readthedocs.io/en/latest/sessions.html