УДК 004

АНАЛИЗ И ВАРИАНТЫ РЕШЕНИЯ ПРОБЛЕМ ОПЕРАТИВНОГО КЭШИРОВАНИЯ ПРИ ИСПОЛЬЗОВАНИИ ТЕХНОЛОГИИ "MEMCACHE"

© 2009 П.В. Назин*

Ключевые слова: кэширование, алгоритмы замещения, оперативная память, PHP, MemcachDB, Berkeley BD, кластеризация.

Рассмотрены вопросы организации кэширования динамически формируемого контента на стороне web-сервера. Особое внимание уделено вопросу надежности функционирования подобных систем, предложены варианты оптимизации.

Сегодня практически любой web-проект является динамическим, т.е. страницы запрашиваемые пользователем формируются "на лету" с учетом индивидуальных особенностей. Для каждого ресурса критичными являются показатели времени отклика сервера, так как увеличение этой характеристики влечет за собой отток пользователей, и как следствие, сокращение времени формирования ответа является одной из приоритетных задач при разработке web-сервисов. Зачастую при формировании страницы необходимо получить данные из относительно медленных источников (например, база данных, удаленный файловый сервер, внешний ресурс) (backend). Для генерации ответа на сложный запрос количество таких обращений может исчисляться десятками, а время обработки каждого обращения варьироваться от десятка миллисекунд до минуты. Даже при параллельной обработке сложных запросов мы получим неудовлетворительное время отклика.

Решением этой задачи является кэширование: мы помещаем результат вычислений в некоторое хранилище, которое обладает отличными характеристиками по времени доступа к информации. Теперь вместо формирования долгих запросов, мы обращаемся к быстрому кэшу.

В случае web-проектов успех кэширования определяется тем, что на сайте есть наиболее популярные страницы, некоторые данные используются на всех или почти на всех страницах, т.е. существуют некоторые выборки, которые оказываются затребованы гораздо чаще других. Поэтому есть возможность заменить несколько обращений к backend'у на запросы к кэшу.

В условиях современного технического обеспечения, наиболее вероятным хранилищем кэша является выделенный массив оперативной памяти сервера, доступный по сетевому протоколу. Технология получила название memcache[1] и представляет собой огромную таблицу хэш-значений. Memcache обеспечивает сервис по хранению значений ассоциированных с ключами. Так как memcache работает с реальным массивом памяти, то алгоритмическая сложность[2] всех его операций должна быть равна O(1), это накладывает сложности связанные с невозможностью даже линейных операций требующих O(n) времени (групповая операция), однако подобная реализация также означает, что скорость работы memcache не зависит от количества ключей которые в нем хранятся. Основными оптимизированными операциями является выделение/освобождение блоков памяти под хранение ключей, определение политики самых неиспользуемых ключей (LRU[3]) для очистки кэша при нехватке памяти. Поиск ключей происходит через хэширование[4], поэтому имеет сложность O(1).

1. Выбор ключей кэширования. Ключом в memcached является строка ограниченной длины, состоящая из ограниченного набора символов. Ключ кэширования должен обладать следующими свойствами:

♦ При изменении параметров выборки, которую мы кэшируем, ключ кэширования должен изменяться (исключение попадания в "чужой" кэш).

* Назин Павел Владимирович, аспирант Поволжского государственного университета телекоммуникаций и информатики. E-mail: vestnik@sseu.ru.

♦ По параметрам выборки ключ должен определяться однозначно, т.е. исключить дублирование записей в кэше.

Самый простейший вариант построения ключа - использовать сроку запроса, например 'user_1' для выборки информации о пользователе с ID=1 или 'friends_2_public_sorted_online' для выборки друзей пользователя с ID 2, которых видно публично. Такой подход чреват ошибками и несоблюдением условий, сформулированных выше (например: в следствии действующих ограничений на размер ключа, обрезанные ключи могут совпадать для нескольких различных запросов).

Можно использовать следующий вариант: если существует некоторая точка в коде, через которую проходят все обращения к БД, а любое обращение полностью описывается (содержит все параметры запроса) в некоторой структуре $options, можно использовать ключ, построенный следующим образом (PHP):

$key = md5(serialize($options)).

Такой ключ несомненно удовлетворяет первому условию (при изменении $options будет обязательно изменен $key), но и второе условие будет соблюдаться, если мы будем все типы данных в $options использовать "канонически", т.е. не допускать строки "1" вместо числа 1 (хотя в PHP два таких значения равны, но их сериализованное представление различается), а функция md5 используется для "сжатия" данных.

2. Утрата ключей. Memcached не является надежным хранилищем - возможна ситуация, когда ключ будет удален из кэша раньше окончания его срока жизни. Архитектура проекта должна гибко реагировать на потерю ключей. Можно выделить три основных причины потери ключей:

1. Ключ был удален раньше окончания его срока годности в силу нехватки памяти под хранение значений других ключей.

2. Ключ был удален, так как истекло его время жизни. Такая ситуация строго говоря не является потерей, так как мы сами ограничили время жизни ключа, однако для frontend'а ситуация будет не удовлетворительной, так как выборка будет формироваться снова.

3. Самой неприятной ситуацией является крах процесса memcached или сервера, на котором он расположен. В этой ситуации мы теряем все ключи, которые хранились в кэше.

Можно обратить внимание, что в memcache удобно хранить не только "горячие" запросы пользователей, но также и текущие сессии пользователей и информацию об их активности, в последнем случае мы рискуем потерять статистику, а пользователям придется заново "логиниться" на сервер, что очень не желательно.

Несколько сгладить последствия позволяет кластерная организация: множество серверов memcache, по которым разнесены ключи проекта: так последствия краха одного кэша будут менее заметны, ключи сессий пользователей в такой схеме можно будет дублировать по физическим серверам кластера и тем самым вовсе избежать потери.

3. Кластеризация и выбор алгоритма распределения ключей по кластерам. Для распределения нагрузки и достижения отказоустойчивости вместо одного сервера memcached используется кластер из таких серверов. Сервера, входящие в кластер, могут быть сконфигурированы с различным объемом памяти, при этом общий объем кэша будет равен сумме объемов кэшей всех memcached, входящих в кластер. Процесс memcached может быть запущен на сервере, где слабо используется процессор и не загружена до предела сеть (например, на файловом сервере). При высокой нагрузке на процессор memcached может не успевать достаточно быстро отвечать на запросы, что приводит к деградации сервиса.

При работе с кластером ключи распределяются по серверам, т.е. каждый сервер обрабатывает часть общего массива ключей проекта. Отказоустойчивость следует из того факта, что в случае отказа одного из серверов ключи будут перераспределены по оставшимся серверам кластера. При этом, конечно же, содержимое отказавшего сервера будет потеряно.

При кластеризации становится актуальным вопрос эффективного распределения ключей по серверам. Для этого необходимо определить функцию распределения ключей, которая по ключу возвращает номер сервера, на котором он должен храниться (или номера серверов, если хранение происходит с избыточностью). Исторически первой фун-

кцией распределения в memcached использовалась функция модуля:

f(ключ) = crc32(ключ)
% количество_серверов.

Такая функция обеспечивает равномерное распределение ключей по серверам, однако проблемы возникают при переконфигурировании кластера mem-cached: изменение количества серверов приводит к перемещению значительной части ключей по серверам, что эквивалентно потере значительной части ключей (так как меняется параметр "количество_серверов").

4. Проблема одновременного перестроения кэшей[6]. Рассмотрим следующую ситуацию: у нас есть выборка из БД, которая используется на многих страницах или особо популярных страницах (например, на главной странице). Эта выборка закэширована с некоторым "сроком годности", т.е. кэш будет сброшен по прошествии некоторого интервала времени. При этом сама выборка является относительно сложной, ее вычисление заметно нагружает backend (БД). В какой-то момент времени ключ в memcached будет удален, так как истечет срок его жизни, в этот момент

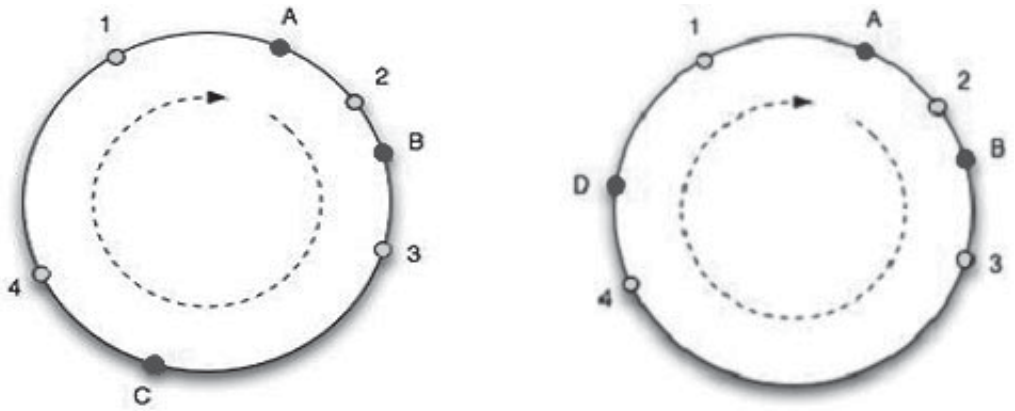

Рис.

Альтернативой для данной функции является механизм консистентного хэширования (consistent hashing[5]), который при переконфигурации кластера сохраняет расположение ключей по серверам.

Суть алгоритма заключается в следующем: мы рассматриваем набор целых чисел от 0 до 2^{32}, "закручивая" числовую ось в кольцо. Каждому серверу из пула memcached-серверов мы сопоставляем число на кольце (см. рисунок, слева сервера A, B и C). Ключ хэшируется в число, в том же диапазоне (на рисунке - точки 1-4), в качестве сервера для хранения ключа мы выбираем сервер в точке, ближайшей к точке ключа в направлении по часовой стрелке. Если сервер удаляется из пула или добавляется в пул, на оси появляется или исчезает точка сервера, в результате чего лишь часть ключей перемещается на другой сервер. На рисунке справа показана ситуация, когда сервер C был удален из пула серверов и добавлен новый сервер D. Легко заметить, что ключи 1 и 2 не поменяли привязки к серверам, а ключи 3 и 4 переместились на другие серверы.

несколько frontend'ов (несколько, так как выборка часто используется) обратятся в memcached по этому ключу, обнаружат его отсутствие и попытаются построить кэш заново, осуществив выборку из БД. То есть в БД одновременно попадет несколько одинаковых запросов, и если выборка не будет совершена, то количество запросов на перестроение будет увеличиваться. В результате сервер БД получит критическую нагрузку и откажется регистрировать новые запросы. Что можно сделать, как избежать такой ситуации?

Проблема с перестроением кэшей становится проблемой только тогда, когда имеют место два фактора: много обращений к кэшу в единицу времени и сложный запрос.

Можно предложить следующую схему: мы больше не ограничиваем время жизни ключа с кэшом в memcached - он будет там находиться до тех пор, пока не будет вытеснен другими ключами. Но вместе с данными кэша мы записываем и *реальное* время его жизни, например:

{годен до: 2009-10-26 11:53,
данные кэша:{...} }

Теперь при получении ключа из memcached мы можем проверить, истек ли срок жизни кэша с помощью поля “годен до”. Если срок жизни истек, кэш надо перестроить, и сделать это с блокировкой, если не удастся заблокироваться, мы можем либо подождать еще (раз блокировка уже есть, значит кэш кто-то перестраивает), либо вернуть старое значение кэша. Все механизмы описанные в данной работе исследовались на стабильном современном модуле кэширования MemcacheDB, представляющего из себя промежуточный интерфейс с ограниченным набором команд между сетевым протоколом и индексной БД (Berkeley DB[7]).

[1] См.: http://memcachedb.org.

[2] См.:http://ru.wikipedia.org/w/index.php?title=Алгоритмическая_сложность.

[3] LRU - алгоритм замещения ключей в кэше, критерием освобождения памяти для которого является возраст замещаемого ключа.

[4] См.: ru.wikipedia.org/wiki/Hash.

[5] См.: http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html.http://www.lastfm.ru/user/RJ/journal/2007/04/10/rz_libketama_-_a_consistent_hashing_algo_for_memcache_clients.

[6] См.: http://korchasa.blogspot.com/2008/04/dog-pile.html.

[7] См.: http://www.oracle.com/technology/products/berkeley-db.

Поступила в редакцию 16.10.2009 г.