Web, кэширование и memcached

Смирнов Андрей (NetStream) / HighLoad++ 2008

Контакты: smira@netstream.ru, http://smira.ru

Оглавление

Введение 1
Memcached и кэширование 2
  Принцип локальности 2
  Memcached 3
  Общая схема кэширования 3
  Архитектура memcached 4
  Потеря ключей 4
Ключ кэширования 5
Кластеризация memcached 6
Атомарность операций в memcached 7
Счетчики в memcached 8
  Счетчик просмотров 8
  Счетчик онлайнеров 9
Проблема одновременного перестроения кэшей 10
  Блокировки в memcached 11
Сброс группы кэшей 12
  Тэг кэша 13
  Версии тэгов 13
Статистика работы memcached 14
  Slab-аллокатор 14
Отладка проектов, использующих memcached 16
Межъязыковое взаимодействие с помощью memcached 16
Материалы 17

Введение

Эта статья была написана на основе материалов одноименного доклада[1] на конференции HighLoad++ (2008). Для начала, о названии статьи: в статье пойдет речь и о кэшировании в Web'е (в высоконагруженных Web-проектах), и о применении memcached для кэширования, и о других применениях memcached в Web-проектах. То есть все три составляющие названия в различных комбинациях будут освещены в этой статье.

Кэширование сегодня является неотъемлемой частью любого Web-проекта, не обязательно высоконагруженного. Для каждого ресурса критичной для пользователя является такая характеристика, как время отклика сервера. Увеличение времени отклика сервера приводит к оттоку посетителей. Следовательно, необхо-

[1] http://www.smira.ru/2008/10/08/highload-plus-plus-2008/

димо минимизировать время отклика: для этого необходимо уменьшать время, требуемое на формирование ответа пользователю, при этом для формирования ответа пользователю необходимо получить данные из каких-то внешних ресурсов (backend). Этими ресурсами могут быть как базы данных, так и любые другие относительно медленные источники данных (например, удаленный файловый сервер, на котором мы уточняем количество свободного места). Для генерации одной страницы достаточно сложного приложения нам может потребоваться совершить десятки подобных обращений. Многие из них будут быстрыми: 20 мс и меньше, однако всегда существует некоторое небольшое количество запросов, время вычисления которых может исчисляться секундами или минутами (даже в самой оптимизированной системе они могут быть, хотя их количество должно быть минимально). Если сложить всё то время, которое мы затратим на ожидание результатов запросов (если же мы будем выполнять запросы параллельно, то возьмем время вычисления самого долгого запроса), мы получим неудовлетворительное время отклика.

Решением этой задачи является кэширование: мы помещаем результат вычислений в некоторое хранилище (например, memcached), которое обладает отличными характеристиками по времени доступа к информации. Теперь вместо обращений к медленным, сложным и тяжелым backend'ам нам достаточно выполнить запрос к быстрому кэшу.

Memcached и кэширование

Принцип локальности

Кэш или подход кэширования мы встречаем повсюду в электронных устройствах, архитектуре программного обеспечения: кэш ЦП (первого и второго уровня), буферы жесткого диска, кэш операционной системы, буфер в автомагнитоле. Чем же определяется такой успех кэширования? Ответ лежит в *принципе локальности*: программе, устройству *свойственно* в определенный промежуток времени работать с некоторым подмножеством данных из общего набора. В случае оперативной памяти это означает, что если программа работает с данными, находящимися по адресу 100, то с большей степенью вероятности следующее обращение будет по адресу 101, 102 и т.п., а не по адресу 10000, например. То же самое с жестким диском: его буфер наполняется данными из областей, соседних по отношению к последним прочитанным секторам, если бы наши программы работали в один момент времени не с некоторым относительно небольшим набором файлов, а со всем содержимым жесткого диска, буфер жесткого диска был бы бессмысленным. Буфер автомагнитолы совершает упреждающее чтение с диска следующих минут музыки потому, что мы, скорее всего, будем слушать музыкальный файл последовательно, а не перескакивать по набору музыки и т.п.

В случае web-проектов успех кэширования определяется тем, что на сайте есть всегда наиболее популярные страницы, некоторые данные используются на всех или почти на всех страницах, то есть существуют некоторые выборки, которые оказываются затребованы гораздо чаще других. Мы заменяем несколько обращений к backend'у на одно обращение для построения кэша, а затем все последующие обращения будем делать через быстро работающий кэш.

Кэш всегда лучше, чем исходный источник данных: кэш ЦП на порядки быстрее оперативной памяти, однако мы не можем сделать оперативную память

такой же быстрой, как кэш – это экономически неэффективно и технически сложно. Буфер жесткого диска удовлетворяет запросы за данными на порядки быстрее самого жесткого диска, однако буфер не обладает свойством запоминать данные при отключении питания – в этом смысле он хуже самого устройства. Аналогичная ситуация и с кэшированием в Web'е: кэш быстрее и эффективнее, чем backend, однако он обычно в случае перезапуска или падения сервера не может сохранить данные, а также не обладает логикой по вычислению каких-либо результатов: он умеет возвращать лишь то, что мы ранее в него положили.

Memcached

Memcached представляет собой огромную хэш-таблицу в оперативной памяти, доступную по сетевому протоколу. Он обеспечивает сервис по хранению значений, ассоциированных с ключами. Доступ к хэшу мы получаем через простой сетевой протокол, клиентом может выступать программа, написанная на произвольном языке программирования (существуют клиенты для C/C++, PHP, Perl, Java и т.п.)

Самые простые операции – получить значение указанного ключа (get), установить значение ключа (set) и удалить ключ (del). Для реализации цепочки атомарных операций (при условии конкурентного доступа к memcached со стороны параллельных процессов) используются дополнительные операции: инкремент/декремент значения ключа (incr/decr), дописать данные к значению ключа в начало или в конец (append/prepend), атомарная связка получения/установки значения (gets/cas) и другие.

Memcached был реализован Брэдом Фитцпатриком (Brad Fitzpatrick) в рамках работы над проектом ЖЖ (LiveJournal). Он использовался для разгрузки базы данных от запросов при отдаче контента страниц. Сегодня memcached нашел своё применение в ядре многих крупных проектов, например, Wikipedia, YouTube, Facebook и другие.

Общая схема кэширования

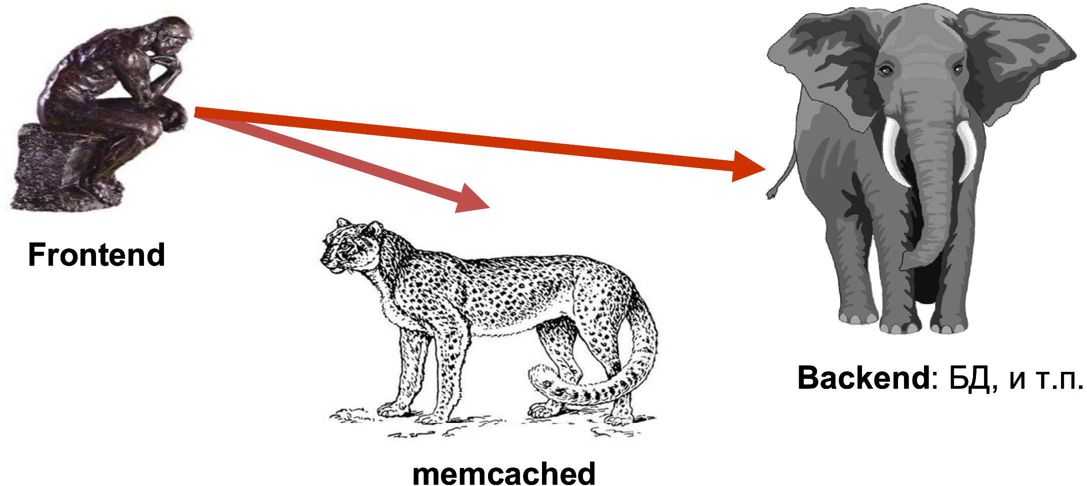


Рисунок 1

В общем случае схема кэширования выглядит следующим образом: frontend'у (той части проекта, которая формирует ответ пользователю) требуется получить данные какой-то выборки. Frontend обращается к быстрому, как гепард, серверу memcached за кэшом выборки (get-запрос). Если соответствующий ключ будет обнаружен, работа на этом заканчивается. В противном случае следует об-

ращение к тяжелому, неповоротливому, но мощному (как слон) backend'у, в роли которого чаще всего выступает база данных. Полученный результат сразу же записывается в memcached в качестве кэша (set-запрос). При этом обычно для ключа задается максимальное время жизни (срок годности), который соответствует моменту сброса кэша.

Такая стандартная схема кэширования реализуется практически всегда. Вместо memcached в некоторых проектах могут использоваться локальные файлы, иные способы хранения (другая БД, кэш PHP-акселератора и т.п.) Однако, как будет показано далее, в высоконагруженном проекте приведенная простейшая схема кэширования может работать не самым эффективным образом. Тем не менее, в нашем дальнейшем рассказе мы будем опираться именно на эту схему как на пример, от которого мы будем отталкиваться.

Архитектура memcached

Каким же образом устроен memcached? Как ему удаётся работать настолько быстро, что даже десятки запросов к memcached, необходимых для обработки одной страницы сайта, не приводят к существенной задержке? Отметим, что memcached крайне нетребователен к вычислительным ресурсам: на нагруженной инсталляции процессорное время, использованное им, редко превышает 10%.

Во-первых, memcached спроектирован так, чтобы все его операции имели алгоритмическую сложность *O(1)*, т.е. время выполнения любой операции не зависит от количества ключей, которые хранит memcached. Это означает, что некоторые операции (или возможности) будут отсутствовать в нём, если их реализация требует всего лишь линейного (*O(n)*) времени. Так, в memcached отсутствуют возможность объединения ключей «в папки», т.е. какой-либо группировки ключей, также мы не найдем групповых операций над ключами или их значениями.

Основными оптимизированными операциями является выделение/освобождение блоков памяти под хранение ключей, определение политики самых неиспользуемых ключей (LRU) для очистки кэша при нехватке памяти. Поиск ключей происходит через хэширование, поэтому имеет сложность O(1).

Используется асинхронный ввод-вывод, не используются нити, что обеспечивает дополнительный прирост производительности и меньшие требования к ресурсам. На самом деле memcached может использовать нити, но это необходимо лишь для использования всех доступных на сервере ядер или процессоров в случае слишком большой нагрузки – на каждое соединение нить не создается в любом случае.

По сути, можно сказать, что время отклика сервера memcached определяется только сетевыми издержками и практически равно времени передачи пакета от frontend'а до сервера memcached (RTT). Такие характеристики позволяют использовать memcached в высоконагруженных web-проектов для решения различных задач, в том числе и для кэширования данных.

Потеря ключей

Memcached не является надежным хранилищем – возможна ситуация, когда ключ будет удален из кэша раньше окончания его срока жизни. Архитектура проекта должна быть готова к такой ситуации и 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. Memcached использует политику LRU, поэтому такая потеря означает, что данный ключ редко использовался и память кэша освобождается для хранения более популярных 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, но для клиентского по отношению к memcached кода такая потеря неотличима от других случаев – при обращении к memcached мы получаем ответ «такого ключа нет».
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. Несколько сгладить последствия позволяет кластерная организация: множество серверов memcached, по которым «размазаны» ключи проекта: так последствия краха одного кэша будут менее заметны.

Все описанные ситуации необходимо иметь в виду при разработке программного обеспечения, работающего с memcached. Можно разделить данные, которые мы храним в memcached, по степени критичности их потери.

«Можно потерять». К этой категории относятся кэши выборок из базы данных. Потеря таких ключей не так страшна, потому что мы можем легко восстановить их значения, обратившись заново к backend'у. Однако частые потери кэшей приводят к излишним обращениям к БД.

«Не хотелось бы потерять». Здесь можно упомянуть счетчики посетителей сайта, просмотров ресурсов и т.п. Хоть и восстановить эти значения иногда напрямую невозможно, но значения этих ключей имеют ограниченный по времени смысл: через несколько минут их значение уже неактуально, и будет рассчитано заново.

«Совсем не должны терять». Memcached удобен для хранения сессий пользователей – все сессии равнодоступны со всех серверов, входящих в кластер frontend'ов. Однако содержимое сессий не хотелось бы терять никогда – иначе пользователей на сайте будет «разлогинивать». Как попытаться избежать? Можно дублировать ключи сессий на нескольких серверах memcached из кластера, так вероятность потери снижается.

Ключ кэширования

Пусть мы уже убедились, что использовать memcached для кэширования – это правильное решение. Первая задача, которая перед нами встаёт – это выбор *ключа* для каждого кэша. Ключом в memcached является строка ограниченной длины, состоящая из ограниченного набора символов (например, запрещены пробелы). Ключ кэширования должен обладать следующими свойствами:

– При изменении параметров выборки, которую мы кэшируем, ключ кэширования должен изменяться (чтобы с новыми параметрами мы не «попали» в старый кэш).
– По параметрам выборки ключ должен определяться однозначно, т.е. для одной и той же выборки ключ кэширования должен быть только один, иначе мы рискуем понизить эффективность процесса кэширования, создавая несколько кэшей для одной и той же выборки.

Конечно, мы могли бы для каждой выборки строить ключ вручную, например ‘user_158’ для выборки информации о пользователе с ID 158 или ‘friends_192_public_sorted_online’ для выборки друзей пользователя с ID 192, которых видно публично и притом отсортированных в порядке последнего появления на сайте. Такой подход чреват ошибками и несоблюдением условий, сформулированных выше.

Можно использовать следующий вариант (пример для PHP): 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: $key = md5(serialize($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). Функция md5 используется для «сжатия» данных: ключ memcached имеет ограничение по длине, а сериализованное представление может быть слишком длинным.

Кластеризация memcached

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, то есть 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 (см. раздел «Потеря ключей»). В случае необходимости важные ключи можно хранить не на одном сервере, а дублировать на нескольких, так можно минимизировать последствия падения сервера за счет избыточности хранения.

При кластеризации становится актуальным вопрос распределения ключей: как наиболее эффективным образом распределить ключи 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 функцией 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 memcached: изменение количества серверов приводит к перемещению значительной части ключей по серверам, что эквивалентно потере значительной части ключей (так как меняется параметр *количество_серверов*).

Альтернативой для данной функции является механизм консистентного хэширования (consistent hashing), который при переконфигурации кластера сохраняет расположение ключей по серверам. Этот подход был реализован в клиентах memcached впервые разработчиками сервиса Last.fm в апреле 2007 года.

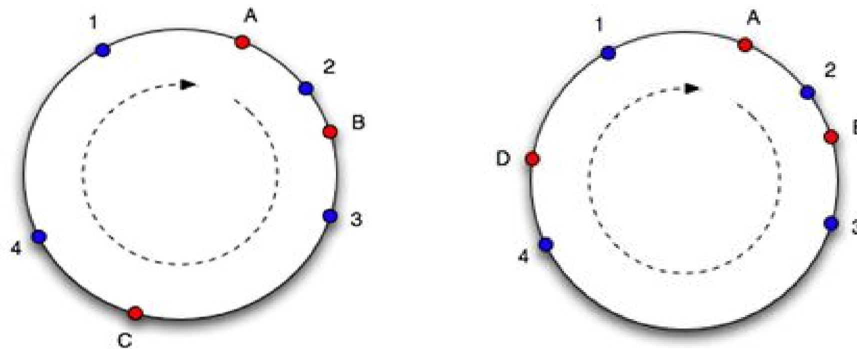


Рисунок 2

Суть алгоритма заключается в следующем: мы рассматриваем набор целых чисел от 0 до 2^{32}, «закручивая» числовую ось в кольцо (склеиваем 0 и 2^{32}). Каждому сервера из пула memcached-серверов мы сопоставляем число на кольце (рисунок 2, слева сервера A, B и C). Ключ хэшируется в число в том же диапазоне (на рисунке – синие 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 2 справа показана ситуация, когда сервер C был удалён 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. На самом деле одному серверу ставится в соответствие 100-200 точек на оси (пропорционально его весу в пуле), что улучшает равномерность распределения ключей по серверам в случае изменения их конфигурации.

Данный алгоритм был реализован во многих клиентах memcached в различных языках программирования, однако реализация иногда отличается деталями, что приводит к несовместимости хэширования. Данный факт делает консистентное хэширование неудобным для использования при доступе к одному пулу серверов memcached из различных языков программирования. Простейший алгоритм с crc32 и модулем реализован во всех клиентах на всех языках одинаково, что обеспечивает одинаковое хэширование ключей по серверам. Поэтому в случае отсутствия необходимости обращаться к memcached из различных клиентов на разных языках программирования более привлекательным выглядит подход с консистентным хэшированием.

Атомарность операций в memcached

Как таковые, все одиночные запросы к memcached атомарны (в силу его однопоточности и корректных внутренних блокировок в многопоточном случае). Это означает, что если мы выполняем запрос get, мы получим значения ключа таким, как кто-то его записал в кэш, но точно не смесь двух записей. Однако каждая операция независима, и мы не можем гарантировать, например, корректность такой процедуры в ситуации конкурентного доступа из нескольких параллельных процессов:

1. Получение значение ключа «x» ($x = get x).
2. Увеличение значения переменной на единицу ($x = $x + 1).
3. Запись нового значения переменной в memcached (set x = $x).

Если данный код выполняют несколько frontend'ов одновременно, может получиться так, что значение ключа x увеличится не n раз, как мы задумывали, а на меньшее значение (классическое состояние гонки, race condition). Конечно, такой подход неприемлем для нас. Классический ответ на сложившуюся ситуацию: применение синхронизационных примитивов (семафоров, мутексов и т.п.), однако в memcached они отсутствуют. Другим вариантом решения задачи является реализация более сложных операций, которые заменяют неатомарную последовательность get/set.

В memcached для решения этой проблемы есть пара операций: incr/decr (инкремент и декремент). Они обеспечивают атомарное увеличение (или, соответственно, уменьшение) целочисленного значения существующего в memcached ключа. Атомарными являются также дополнительные операции: append/prepend, которые позволяют добавить к значению ключа данные в начало или в конец, также в каком-то плане атомарными можно считать операции add и replace, которые позволяют задать значение ключа, только если он ранее не существовал, или, наоборот, заменить значение уже существующего ключа.

Необходимо дополнительно отметить, что любая блокировка в memcached должна быть мелкозернистой (fine-grained), то есть должна затрагивать как можно меньшее число объектов, так как основная задача сервера – обеспечивать эффективный доступ к кэшу как можно большего числа параллельных процессов.

Счетчики в memcached

Memcached может использоваться не только для хранения кэшей выборок из backend'ов, не только для хранения сессий пользователей (о чем было упомянуто в начале статьи), но и для задачи, которая без memcached решается достаточно тяжело, – реализация счетчиков, работающих в реальном времени. Т.е перед нами стоит задача показывать текущее значение счетчика в данный момент времени, если откинуть требование «реального времени», это можно реализовать через логирование и последующий анализ накопленных логов.

Рассмотрим несколько примеров таких счетчиков, как их можно реализовать, какие возможны проблемы.

Счетчик просмотров

Пусть в нашем проекте есть некоторые объекты (например, фото, видео, статьи и т.п.), для которых мы должны в реальном времени показывать число просмотров. Счетчик должен увеличиваться с каждым просмотром. Самый простой вариант – при каждом просмотре обновлять поле в БД, – не будет работать, т.к. просмотров много и БД не выдержит такую нагрузку. Мы можем реализовать точный и аккуратный сбор статистики просмотров, их аккумулирование, и периодический анализ, который заканчивается обновлением счетчика в базе данных (например, раз в час). Однако остается задача показа текущего количества просмотров.

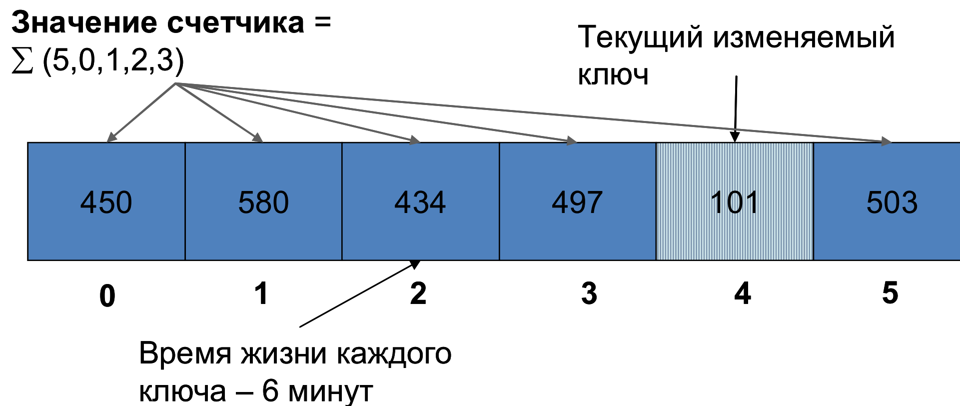
Рассмотрим следующее возможное решение. Frontend в момент просмотра объекта формирует имя ключа счетчика в memcached, и пытается выполнить операцию incr (инкремент) над этим ключом. Если выполнение было успешным,

это означает, что соответствующий ключ находится в memcached, мы просмотр засчитали, также мы получили новое значение счетчика (как результат операции incr), которое мы можем показать пользователю. Если же операция incr вернула ошибку, то ключ счетчика в данный момент отсутствует в memcached, мы можем выбрать в качестве начального значения число просмотров из базы данных, увеличить его на единицу, и выполнить операцию set, устанавливая новое значение счетчика. При последующих просмотрах ключ уже будет находиться в memcached, и мы будем просто увеличивать его значение с помощью incr.

Необходимо отметить, что приведенная схема не является вполне корректной: в ней присутствует состояние гонки (race condition). Если два frontend одновременно обращаются к счетчику, одновременно обнаруживают его отсутствие, и сделают две операции set, мы потеряем один просмотр. Это можно считать не очень критичным, так как процесс аккумулирования статистики восстановит правильное значение. В случае необходимости можно воспользоваться блокировками в memcached, речь о которых пойдет ниже. А можно операцию set заменить на add, проверяя ошибочную ситуацию о существовании ключа.

Счетчик онлайнеров

Существует еще один вид счетчиков, который без memcached или подобного ему решения вряд ли может быть реализован: это счетчик «онлайнеров». Такие счетчики мы видим на большом количестве сайтов, однако в первую очередь необходимо определить, что же именно мы имеем в виду под «онлайнером». Пусть мы хотим рассчитать, сколько уникальных сессий (пользователей) обратилось к нашему сайту за последние 5 минут. Уникальность обращения пользователя с данной сессией в течение 5 минут можно отследить, сохраняя в сессии время последнего засчитанного обращения, если прошло более 5 минут – значит это новое (уникальное) обращение.



Онлайнеры – кол-во уникальных сессий за последние 5 минут

Рисунок 3

Итак, выделим в memcached шесть ключей с именами, например, c_0, c_1, c_2, …, c_5. Текущим изменяемым ключом мы будем считать ключ с номером, равным остатку от деления текущей минуты на 6 (на рисунке это ключ c_4). Именно его мы будем увеличивать с помощью операции incr при каждом обращении уникальной в течение 5 минут сессии. Если incr вернет ошибку (ключа еще нет), установим его значение в 1 с помощью set, обязательно указав время жизни 6 минут. Значением счетчика онлайнеров будем считать сумму всех ключей, кроме текущего (на рисунке это ключи c_0, c_1, c_2, c_3 и c_5).

Когда наступит следующая минута, текущим изменяемым ключом станет ключ c_5, при этом его предыдущее значение исчезнет (т.к. он был создан 6 минут назад с временем жизни те же 6 минут). Значением счетчика станет сумма ключей с c_0 по c_4, т.е. только что рассчитанное значение ключа c_4 уже начнет учитываться в отображаемом значении счетчика.

Такой счетчик может быть построен и на меньшем числе ключей. Минимально возможными для данной схемы являются два ключа: один обновляется, значение другого показывается, затем по прошествии 5 минут счетчики меняются местами, при этом тот, который только что обновлялся, сбрасывается. В приведенной схеме с многими ключами обеспечивается некоторое «сглаживание», которое обеспечивает более плавное изменение счетчика в случае резкого притока или оттока посетителей.

Проблема одновременного перестроения кэшей

Данная проблема характерна в первую очередь для высоконагруженных проектов. 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, её вычисление заметно нагружает backend (БД). В какой-то момент времени ключ в memcached будет удален, т.к. истечет срок его жизни (срок жизни был установлен у кэша), в этот момент несколько frontend'ов (несколько, т.к. 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, каждый из которых заметно нагружает базу данных, при превышении некоторого порога запрос не будет выполнен за разумное время, еще больше frontend'ов обратятся к кэшу, обнаружат его отсутствие и отправят еще больше запросов в базу данных, с которыми база данных тем более не справится. В результате сервер БД получит критическую нагрузку, и «ляжет». Такая ситуация называется dog-pile effect[2]. 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. Причем один фактор может компенсировать другой: на относительно непопулярной, но очень сложной и долгой выборке (которых вообще-то не должно быть) мы можем получить аналогичную ситуацию. Итак, что же делать?

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: 2008-11-03 11:53,
    данные кэша:
    {
```

[2] См. также: http://korchasa.blogspot.com/2008/04/dog-pile.html, http://blog.kovyrin.net/2008/03/10/dog-pile-effect-and-how-to-avoid-it-with-ruby-on-rails-memcache-client-patch/

```
        ...
    }
}
```

Теперь при получении ключа из memcached мы можем проверить, истёк ли срок жизни кэша с помощью поля «годен до». Если срок жизни истёк, кэш надо перестроить, но мы будем делать это с *блокировкой* (о блокировках речь пойдет в следующем разделе), если не удастся заблокироваться, мы можем либо подождать еще (раз блокировка уже есть, значит кэш кто-то перестраивает), либо вернуть старое значение кэша. Если заблокироваться удастся, мы строим кэш самостоятельно, при этом другие frontend'ы не будут перестраивать этот же кэш, так как увидят нашу блокировку. Основное преимущество хранения кэша в memcached без указания срока годности – именно возможность получить его старое значение в случае, если кэш уже перестраивается кем-то. Что именно делать – ждать, пока кэш построит кто-то другой и получать новое значение из memcached или возвращать старое значение, – зависит от задачи, насколько приемлемо старое значение и сколько можно провести времени в состоянии ожидания. Чаще всего можно позволить себе 2-3 секундное ожидание с проверкой удаления блокировки и, если кэш так и не построился (что маловероятно, получается что выборка происходит больше чем за 2-3 секунды), вернуть старое значение, освобождая frontend для других задач.

Пример такого алгоритма:

```
Получаем доступ к кэшу cache, его срок жизни истёк.
Пытаемся заблокироваться по ключу user_cache_lock.

    Не удалось получить блокировку:
       ждём снятия блокировки;
       не дождались: возвращаем старые данные кэша;
       дождались: выбираем значения ключа заново, возвращаем новые данные
(построенный кэш другим процессом).

    Удалось получить блокировку:
       строим кэш самостоятельно.
```

Такая схема позволяет исключить или свести к минимуму ситуации «заваливания» backend'а одинаковыми «тяжелыми» запросами, когда реально запрос достаточно выполнить лишь один раз. Остается последний вопрос, как обеспечить корректную блокировку? Очевидно, что так как проблема одновременного перестроения возникает на разных frontend'ах, то блокировка должна быть в общедоступном для них всех месте, то есть в memcached.

Блокировки в memcached

Рассмотрим два варианта реализации блокировки (мьютекса, двоичного семафора) с помощью memcached. Первый некорректный, он не может обеспечить корректного исключения параллельных процессов, но очевидный. Второй совершенно корректный, но не настолько очевиден.

Пусть мы хотим заблокироваться по ключу 'lock'. Делаем попытку получить значение ключа с помощью операции get. Если ключ не найден, значит блокировки нет, и мы с помощью операции set устанавливаем значение этого ключа, например, в единицу, а время жизни устанавливаем в небольшой интервал времени, который превышает максимальное время жизни блокировки, например, в 10 секунд. Теперь, если frontend завершится аварийно и не снимет блокировку, она автоматически уничтожится через 10 секунд. Итак, с помощью set мы блоки-

ровку установили, выполнили все необходимые действия, после этого снимаем блокировку, просто удаляя соответствующий ключ командой del. Если на первой операции get мы получили значение ключа, это означает, что блокировка уже установлена другим процессом, наша операция блокировки неуспешна.

Описанный способ обладает недостатком: наличием состояния гонки (race condition). Два процесса могут одновременно сделать get, оба могут получить ответ, что «ключа нет», оба сделают set, и оба будут считать, что установили блокировку успешно. В таких ситуациях, как одновременное перестроение кэшей, это может быть допустимо, т.к. здесь цель не исключить все другие процессы, а резко уменьшить количество одновременных запросов к БД, что может обеспечить и этот простой, некорректный вариант.

Второй вариант корректен, и даже проще первого. Для захвата блокировки достаточно выполнить одну команду: add, указав имя ключа и время жизни (такое же маленькое, как и в первом варианте). Команда add будет успешной только в том случае, если ключа в memcached еще нет, то есть наш процесс и есть тот единственный процесс, которому удалось захватить блокировку. Тогда нам надо выполнить необходимые действия и освободить блокировку командой del. Если add вернет ошибку «такой ключ уже существует», значит, блокировка была захвачена раньше каким-то другим процессом.

Сброс группы кэшей

Если мы закэшировали какие-то данные от backend'a, например, выборку из БД, рано или поздно исходные данные изменятся, и кэш перестает быть валидным. Причем очень желательно, чтобы кэш сбрасывался сразу же за изменением, иначе пользователь после редактирования может увидеть старую версию объекта, что его, несомненно, смутит. Есть простой вариант ситуации: мы меняем информацию об объекте с ID 35, и сбрасываем кэш выборки этого объекта по параметру ID=35. На практике же чаще всего один и тот же объект явно или неявно входит в большое количество выборок, а значит и кэшей.

Рассмотрим такой пример: мы написали блогохостинг, в нем большое количество блогов. Когда один из авторов создает новый пост, меняется большое количество выборок: посты на главной странице и всех вторых страницах списка постов (т.к. все посты «сдвинулись» на один), изменилось количество записей в календаре постов, изменилась RSS-ка, и т.п. Конечно, мы могли бы поставить кэшам этих выборок небольшое время жизни, тогда через какое-то время они сбросятся и будут отображать правильную информацию, но слишком короткое время кэширования (5 секунд, например), будет давать низкое соотношение хитов в кэш, увеличивая нагрузку на БД, а более длительное будет создавать у пользователя ощущение, что информация после создания поста не обновилась, а, значит, пост не добавился. В то же время можно заметить, что в рамках блогохостинга если даже мы сбросим все кэши, связанные с данным блогом, это совсем небольшой процент от общей массы кэширования (т.к. блогов очень много). Остался вопрос: как найти и проидентифицировать все кэши данного блога? Какие-то из них мы можем легко построить, для некоторых это становится уже неудобно: например, количество кэшей постраничного списка постов зависит от количества страниц, которое еще необходимо вычислить. Что же делать?

Одно из возможных решений – тэгирование кэшей. Описанный ниже способ тэгирования по своей сути совпадает с описанным Дмитрием Котеровым в его

наблах[3], но был нами разработан независимо. Существуют и другие варианты тэгирования, например, патч memcached-tag[4] на memcached.

Тэг кэша

Итак, мы вводим новое понятие – тэг кэша. Кэш связан с некоторым списком тэгов. Сам по себе тэг – это некоторое имя и связанная с ним версия (число). Версия тэга может только монотонно увеличиваться. Группой кэшей мы будем называть кэши, имеющие один общий тэг. Для того чтобы сбросить группу кэшей, достаточно увеличить версию соответствующего тэга.

На программном уровне мы знаем, что данная выборка должна быть закэширована и что её кэш будет связан с тэгами tag1 и tag2 (данный факт определяется логикой работы нашего приложения). При создании кэша мы записываем в него кроме данных закэшированной выборки еще текущие (на момент создания кэша) версии тэгов tag1 и tag2. При получении кэша мы считаем его валидным, если не истекло время его жизни, и при этом текущии версии тэгов tag1 и tag2 равны версиям, записанным в кэше. Таким образом, если мы изменяем (увеличиваем) версию тэга tag1, все кэши, связанные с этим тэгом, которые были построены ранее, перестанут быть валидными (т.к. в них записана меньшая версия тэга tag1).

Рассмотрим пример с нашей выборкой, пусть было так:

```
Версии тэгов:
  tag1 → 25
  tag2 → 63

Кэш выборки:
  [
    срок годности: 2008-11-07 21:00
    данные кэша: [
                    …
                  ]
    тэги: [
          tag1: 25
          tag2: 63
           ]
   ]
```

Затем произошло некоторое событие, и мы решили сбросить все кэши, ассоциированные с тэгом tag2, т.е. мы увеличили версию тэга: tag2++. Изменились версии тэгов:

```
Версии тэгов:
  tag1 → 25
  tag2 → 64
```

Теперь наш кэш перестал быть валидным, не смотря на то, что его «срок годности» еще не истёк: версия тэга tag2, сохраненная в нем (63) не совпадает с текущей версией (64).

Версии тэгов

Тэги (то есть их версии) имеет смысл хранить там же, где мы храним и наши кэши, то есть в memcached. Для каждого тэга мы создаём ключ с именем, сов-

[3] http://dklab.ru/chicken/nablas/47.html
[4] http://code.google.com/p/memcached-tag/

падающим с именем тэга, его значением будет версия тэга. Осталось решить, что использовать в качестве версии тэга? Можно было бы использовать просто числа, инкрементируя их при изменении версии тэга, но это может привести к некорректному поведению при условии возможной потери ключей. Пусть версия тэга равнялась единице, мы закэшировали выборку с этим тэгом, записали в кэш значение тэга – единицу. Затем ключ с версией тэга был удален из memcached, а в следующий момент времени мы захотели сбросить выборки, связанные с тэгом, то есть необходимо увеличить версию тэга. Так как мы потеряли значение версии тэга, мы снова поставим единицу, и теперь наш кэш будет считаться валидным, хотя он сбросился (не важно, какое значение выбирать при увеличении версии тэга, если она была потеряна – всегда возможна ситуация, что это же значение использовалось и ранее).

В качестве версии удобнее использовать текущее время (с достаточной точностью, например, до миллисекунд). Тогда увеличение версии тэга будет всегда давать новую, бóльшую версию, даже в случае потери предыдущей версии. Версия тэга формируется на frontend'ах, их системные часы должны быть синхронизованы (без этого не будет работать и другая функциональность, например, корректное вычисление срока годности кэшей с коротким временем жизни), так что проблем с таким выбором способа вычисления версии не должно быть.

Использование текущего времени в качестве версии тэга даёт еще одно преимущество в ситуации, когда БД проекта устроена по схеме мастер-слейв репликации. При изменении исходного объекта в БД мы изменяем версию тэга, связанного с ним (записываем туда текущее время, то есть время изменения). В другом процессе мы обнаруживаем, что кэш устарел, то есть его надо перестроить, перестроение – это читающий запрос (SELECT), который необходимо отправить на слейв-сервер БД, но в силу задержек репликации слейв-сервер еще мог не получить актуальную версию объекта в БД, в результате мы кэш сбросили, но при его перестроении снова закэшировали старый вариант объекта, что неприемлемо. Можно использовать версию тэга при решении вопроса, на какой сервер БД отправить запрос: если разница между текущим временем и версией какого-либо тэга кэша меньше некоторого интервала, определяемого максимальной задержкой репликации, мы отправляем запрос на мастер-сервер БД вместо слейва.

Использование такой схемы тэгирования увеличивает количество запросов к memcached, т.к. нам необходимо для каждого кэша получать версии его тэгов. Накладные расходы можно сократить за счет использование multi-get запросов memcached, а также за счет локального кэширования ключей memcached в пределах одного процесса (если один и тот же тэг привязан к нескольким кэшам).

Статистика работы memcached

Кроме необходимости реализовать механизмы работы с memcached, необходимо постоянно заниматься мониторингом кластера memcached-серверов, чтобы быть уверенным, что мы достигли оптимальной производительности. Memcached предоставляет набор команд для получения информации о его работе.

Самая простая команда, stats, позволяет получить элементарную статистику: время работы сервера (uptime), объем используемой памяти, количество get запросов и количество хитов (hits), т.е. попаданий в кэш. Их соотношение позволяет нам судить об эффективности кэширования в целом, хотя необходимо учитывать, что в memcached ключами являются не только закэшированные выборки,

но и счетчики, блокировки, тэги и т.п., так что для вычисления чистой эффективности кэширования это значение требует корректировки. Из общей статистики мы также можем узнать, сколько ключей было удалено раньше истечения срока жизни (evictions), данный параметр может сигнализировать о недостаточности объема памяти memcached.

Slab-аллокатор

Для распределения памяти под значения ключей memcached использует вариант slab-аллокатора[5]. Данный тип аллокатора стремится сократить внутреннюю фрагментацию при выделении памяти, а также обеспечивают хорошую эффективность операций выделения памяти.

Механизм его работы заключается в том, что вся доступная memcached память делится на slab'ы (блоки), каждый из которых будет хранить элементы определенного размера. Например, slab для хранения объектов размером 256 байт, при этом сам slab имеет размер 1 Мб, таким образом он может сохранить 4096 таких объектов. Память внутри такого slab'a выделяется только по 256 байт. Если у нас есть slab'ы для объектов размером 64, 128, 256, 1024 и 2048 байт, то максимальный размер объекта, который мы можем сохранить – 2048 байт (в последнем slabe). Если мы хотим сохранить объект размером 65 байт, под него будет выделена память в slab'e-128, 1 байт – в slab'e 64.

Чтобы добиться эффективного использования памяти memcached для хранения наших ключей и значений, мы должны быть уверены в правильном выборе размеров slab'ов, который выделил memcached, а также в их разумном наполнении. Для этого мы можем попросить memcached предоставить статистику по slab'ам, которую можно, например, визуализировать в виде такого графика:

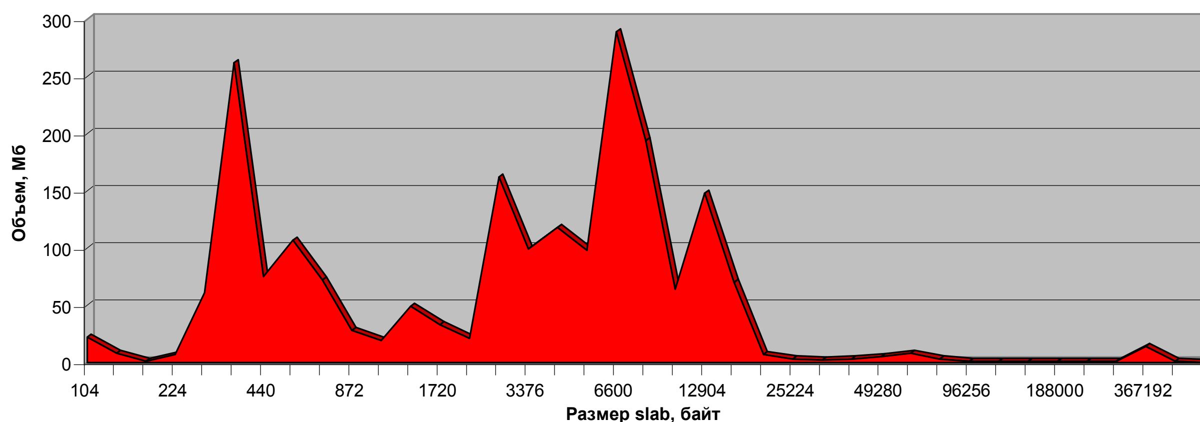


Рисунок 4

Здесь на горизонтальной оси отложены размеры slab'ов, а на вертикальной – объем памяти, используемый slab'ами данного размера. В данный момент вся память memcached занята ключами и их значениями, поэтому данный график представляет собой текущее распределение значений в памяти сервера. Легко видеть, что больше всего slab'ов выделено под ключи с относительно небольшими значениями – до 20 Кб, для больших по размеру ключей slab'ов гораздо меньше. Такое распределение адекватно нашей задаче: у нас больше всего именно маленьких ключей (счетчики, блокировки, небольшие кэши). При этом эти же ключи занимают и бóльшую часть памяти, с локальными пиками выделения под ключи

[5] http://en.wikipedia.org/wiki/Slab_allocation

размером 300 байт, 8 Кб. Если график отличается от того, который ожидается по логике задачи, это повод для беспокойства.

Отладка проектов, использующих memcached

Мы написали большую подсистему для работы с memcached, реализовали различные механизмы решения проблем, связанных с высокой нагрузкой. Как проверить, что всё действительно работает так, как нам бы этого хотелось? Высокую нагрузку, сетевые задержки и т.п. практически невозможно воспроизвести в локальном окружении, непросто это сделать и в тестовом окружении. На серверах в production нам доступны лишь те механизмы отладки, которые не затрагивают нормальное функционирование самого приложения. Способ отладки не должен вносить ощутимых временных задержек, иначе он изменит поведение приложения, и отладка станет бессмысленной.

Можно предложить следующий «трюк», который может помочь в данной ситуации: для каждого кэша (ключа в memcached) или для группы кэшей (ключей) мы заводим отдельный файл в локальной файловой системе. В этот файл в режиме append мы дописываем по одному символу в ответ на каждое логическое действие, которое произошло с кэшом. Для просмотра в реальном времени поведения кэширующей подсистемы достаточно сделать tail –f на этот файл:

```
MLWUHHHHHHHHHHHHHHHHMLLHHHHHHHHHH
```

Пусть буквы имеют следующий смысл:

- M – кэш устарел (или не найден);
- L – попытка заблокироваться;
- W – запись (и построение) нового кэша;
- U – удаление блокировки;
- H – успешный запрос кэша.

Тогда по приведенной последовательности можно рассказать то, что происходило с данным кэшом: вначале он отсутствовал, мы кэш не обнаружили (M), попытались заблокироваться (L) для его построения, заблокировались, построили кэш (W), сняли блокировку (U), затем какое-то время кэш успешно работал, отдавая закэшированные данные (H). Потом в какой-то момент кэш устарел или был сброшен (M), мы попытались заблокироваться, не получилось (L), попытались еще раз (L), блокировка оказалась снята, кто-то другой построил новый кэш, мы его прочитали (H) и дальше им пользовались.

Межпроцессное взаимодействие с помощью memcached

Сложный проект состоит из отдельных компонент, сервисов, которые должны взаимодействовать друг с другом, используя механизмы RPC, вызовы API, обмениваясь информацией через БД или каким-то еще способом. Иногда для такого обмена информацией можно использовать и memcached.

В качестве примера рассмотрим сервис пользовательских вещаний: существует какое-то количество вещаний, в каждом из которых в данный момент времени находится некоторое количество зрителей. Популярность вещания определяется количеством зрителей. Актуальной информацией о количестве зрителей обладает только сервер вещаний, а список вещаний на странице вещаний формирует frontend. Конечно, можно было бы сделать так, чтобы сервер вещаний периодически сбрасывал в БД или через API в frontend информацию о количестве

зрителей, или frontend мог бы через API сервера вещаний получать актуальную информацию. Однако количество зрителей – очень быстро меняющаяся характеристика, и в данной ситуации можно просто из сервера вещаний периодически (раз в несколько секунд) сохранять в memcached информацию о количестве зрителей в каждом из вещаний, а frontend, обращаясь к memcached, может получить информацию в любой удобный момент. Таким может быть межпроцессное взаимодействие, реализованное с помощью memcached.

Материалы

1. http://danga.com/memcached/
2. http://en.wikipedia.org/wiki/Memcached
3. http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html
4. http://www.lastfm.ru/user/RJ/journal/2007/04/10/rz_libketama_-_a_consistent_hashing_algo_for_memcache_clients
5. http://ru.wikipedia.org/wiki/Состояние_гонки
6. http://dklab.ru/chicken/nablas/47.html
7. http://code.google.com/p/memcached-tag/
8. http://korchasa.blogspot.com/2008/04/dog-pile.html
9. http://blog.kovyrin.net/2008/03/10/dog-pile-effect-and-how-to-avoid-it-with-ruby-on-rails-memcache-client-patch/