HighLoad++

Web, кэширование и memcached

Андрей Смирнов (НетСтрим)

Кэширование

- **Время отклика сервера** – важный фактор для пользователей.
- Для сложного сайта генерация одной страницы ~ 20-50 запросов к БД.
- Вычислительно сложные задачи (запросы) ~ 1-∞ секунд.
- Кэширование как способ минимизации времени отклика и снижения нагрузки на сервер.

Кэш

- Кэш встречается везде: ЦП, жесткий диск, магнитола в машине, буферы ОС, …
- Успех кэша в **принципе локальности**.

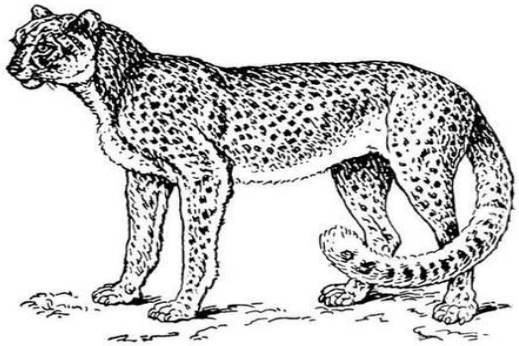
memcached

- Большая хэш-таблица в памяти, доступная через сетевой протокол.
- Операции:
  - get/set/del
  - «Атомарность»
    - incr/decr
    - cas/add/replace
    - append/prepend

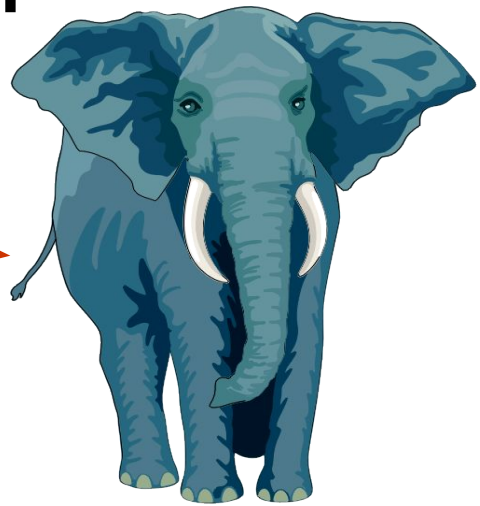
Brad Fitzpatrick

Общая схема кэширования

Frontend



memcached



Backend: БД, и т.п.

Архитектура memcached

- Никаких вычислительно сложных операций.
- Все операции – **O(1)**.
- Никаких нитей – асинхронный ввод/вывод.
- Время отклика сервера – почти RTT.

Потеря ключей

- Ограниченность объема памяти, выделенного memcached.
- Истек срок жизни ключа.
- Отказ сервера или процесса memcached.

Применение memcached

- «Можно потерять»:
  - кэширование выборок БД;
  - вычислительно сложные значения.
- «Не хотелось бы потерять»:
  - счетчики посетителей, просмотров и т.п.
- «Совсем не должны терять»:
  - сессии пользователей.

Задачи

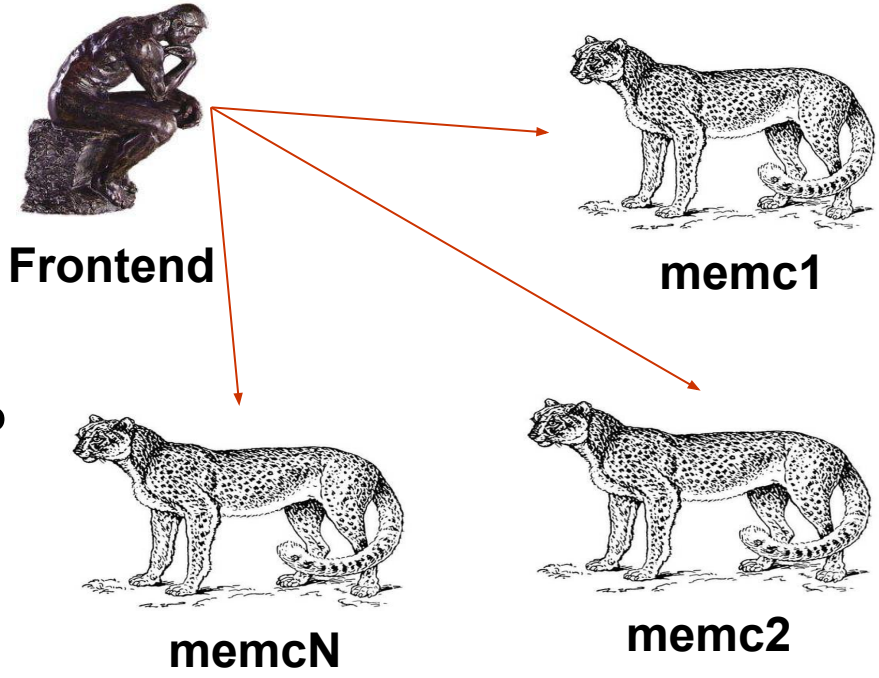
1. Формирование ключа кэширования.
2. Кластеризация memcached.
3. Счетчики и атомарность.
4. Как избежать одновременного перестроения кэшей.
5. Сброс группы кэшей.
6. Анализ статистики memcached, slab-аллокатор.
7. Отладка memcached, дополнительные вопросы.

Ключ кэширования

- **Ключ** – строка ограниченной длины.
  - По параметрам выборки должен однозначно определяться ключ.
  - При изменении параметров выборки ключ должен изменяться.
- Вариант: `ключ = md5(serialize(параметры))`

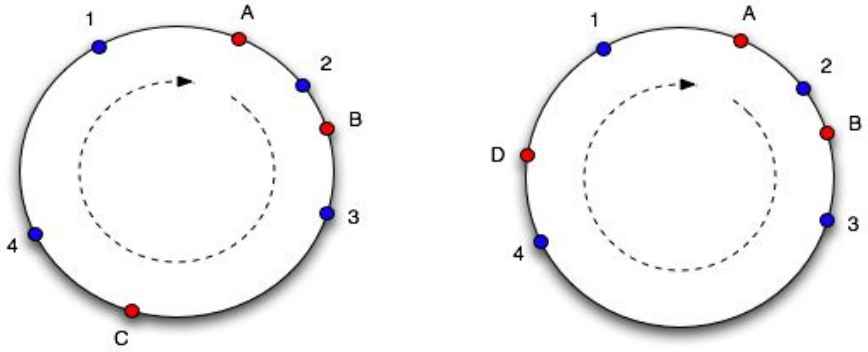
Кластеризация memcached

- Зачем:
  - увеличение объема кэша;
  - обеспечение некоторой отказоустойчивости;
  - распределение нагрузки.
- Как распределить ключи?



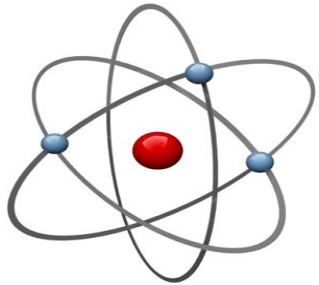
Распределение ключей

- Необходима функция: `f(ключ)=номер_сервера`
- «Стандартный вариант» по модулю:
  `f(ключ)=crc32(ключ)%кол-во_серверов`
- Consistent hashing:



Атомарность операций

- memcached не обеспечивает операций блокировки.
- Обычные операции get/set не обеспечивают атомарности.
- Самые простые атомарные операции: инкремент/декремент (incr/decr).



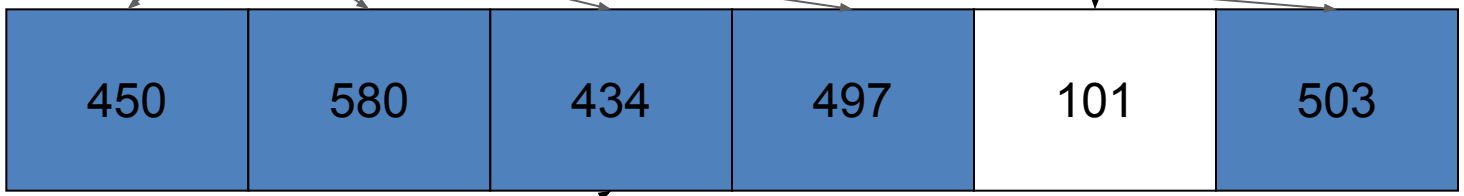
Счетчики в memcached

- Пример: счетчик просмотров в реальном времени.
  - число просмотров аккумулируется и сохраняется в БД;
  - после просмотра увеличиваем (incr) счетчик в memcached;
  - если получили ошибку, выбираем начальное значение из БД (set).
- Наличие race condition.

Счетчик онлайнеров

Значение счетчика =
∑ (5,0,1,2,3)

Текущий изменяемый ключ



Время жизни каждого ключа – 5 минут

Онлайнеры – кол-во уникальных сессий за последние 5 минут

Одновременное перестроение кэшей

- Пусть есть кэш с **большим** количеством обращений на чтение.
- В какой-то момент истекает срок жизни кэша.
- Большое число frontendов пытаются **одновременно** перестроить кеш.
- Получаем **огромную** нагрузку на backend в один момент времени.

Решение проблемы

- Храним ключи кэшей без ограничения по времени.
- В значение кэша записываем реальное время жизни кэша.
- Если получили **устаревший** кэш, предпринимаем попытку перестроения с **блокировкой**.
- **Если** кто-то **уже** перестраивает кэш, подождем или **вернём старое значение**.

Пример

1. Обращаемся за кэшем, например ‘user_info_id_159’
2. Сравниваем срок годности с текущим временем.
3. Кэш «протух» → необходимо его постро заново.

Ключ user_info_id_159:

срок годности:
    2008-10-07 21:00
данные кэша: [
    id: 159
    login: ‘user’
    nick: ‘Hello’
    …
]

Пример

- Пытаемся заблокироваться по ключу user_info_id_159_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.

Блокировки в memcached

- Первый вариант: get/set блокировка
  - get(lock) ? 1 → locked
  - set(lock, 1, small_timeout) … delete(lock)
  - неатомарная, простая, работоспособна для нас.
- Корректная блокировка: gets/cas блокировка
  - gets(lock) → значение, unique
  - cas(lock, 1, unique, small_timeout)
  - атомарна, корректна.

Сброс группы кэшей

- Один и тот же объект часто входит в несколько разных выборок, а значит и кэшей, т.е. изменение объекта должно приводить к инвалидации группы кэшей.
- memcached не поддерживает «папки», т.к. это противоречит сложности O(1) для всех операций.
- **Что делать?**

Тэгирование кэшей

- Тэг – это имя и версия группы кэшей.
- Версия – монотонно увеличивающееся число.
- Сброс группы кэшей – увеличение версии тэга группы.

Тэгирование кэшей

- В memcached вместе с данными кэша отправляем **номера версий всех тэгов**, которые были актуальны на момент создания кэша.
- При получении кэша, он считается валидным, если:
  - у него не истекло собственное «время жизни»;
  - текущая версия всех тэгов, с которыми связан кэш, равна версиям, записанным в кэше.

Пример

Было:

tag1 → 25

tag2 → 63

Записали в кэш:

срок годности: 2008-10-07 21:00
данные кэша: […
]
тэги: [
tag1 : 25
tag2 : 63
]

Пример

tag2++

Пример

Стало:

tag1 → 25

tag2 → 64

Лежит в кэше:

срок годности: 2008-10-07 21:00
данные кэша: […
]
тэги: [
tag1 : 25
tag2 : 63
]

кэш устарел!

Версия тэга и слейвы БД

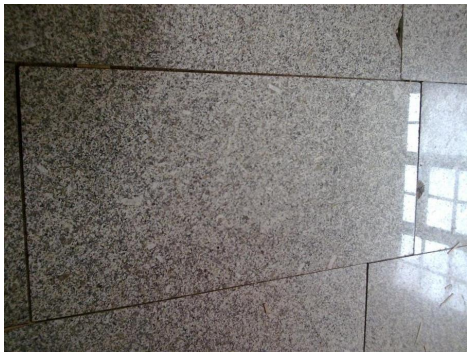
- Удачный вариант версии – текущее время:
  - монотонно увеличивается;
  - при потере значения тэга в memcached корректно восстанавливается.
- Версия в виде времени может использоваться для компенсации задержки в синхронизации слейвов БД:
  - если (текущее время – версия) < 10 сек., используем для выборки мастера.

Статистика memcached

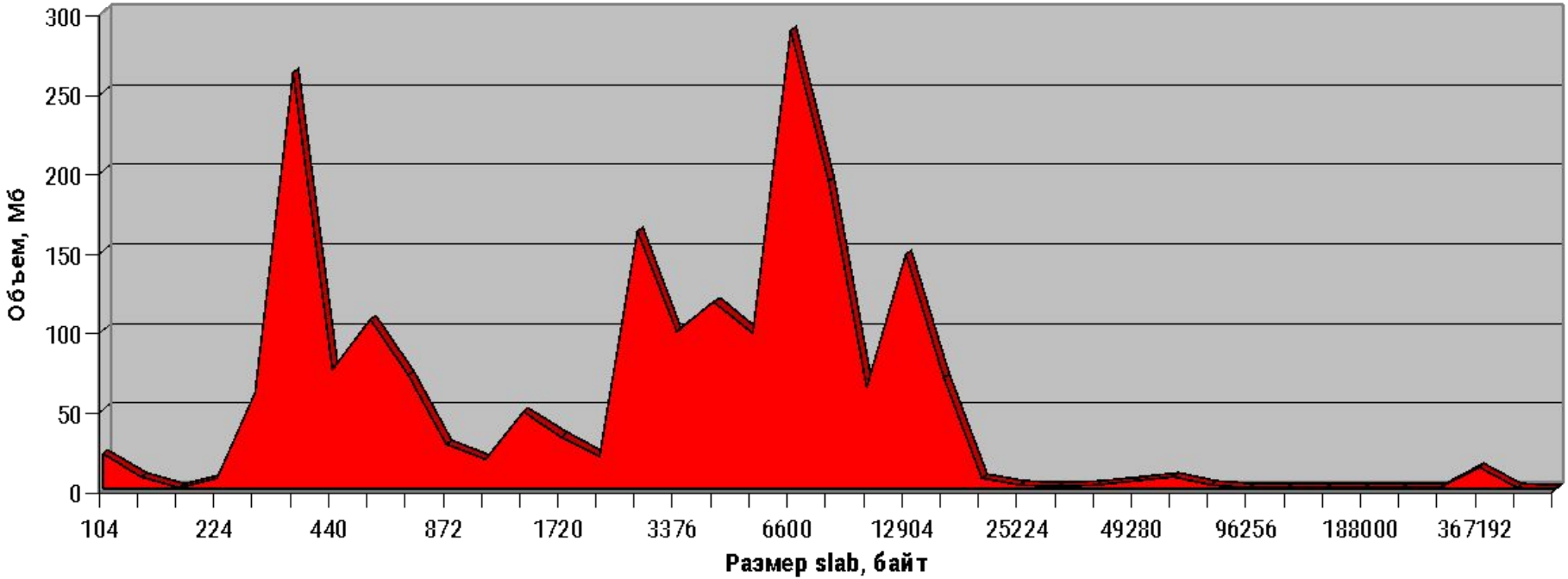
- Команда **stats** позволяет получить различную статистику по работе memcached.
- «Обычная статистика»:
  - процент хитов по отношению к общему числу «get» (эффективность кэша);
  - ключи, удаленные раньше времени из кэша (достаточность объема памяти);
  - объем памяти процесса, uptime и т.п.

slab-аллокатор

- Баланс между внутренней фрагментацией и эффективностью использования памяти.
- Эффективные O(1) алгоритмы.
- Набор slab'ов под блоки предопределенного размера: 64, 128, 256, …, 2^{10}.
- Каждый slab: использовано кусков, занято кусков, список свободных кусков, очередь LRU.



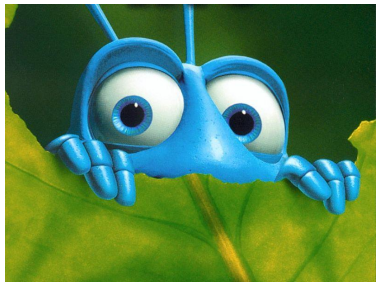
Статистика slab-аллокатора



Отладка memcached

- Проблемы плохо воспроизводятся в локальном/тестовом окружении.
- Отладка возможна только в реальном времени (без остановок).
- Вариант решения: одно действие – один символ в лог:

```
MLWUHHHHHHHHHHHHHHHMLLHHHHHHHHHH
```



Дополнительные вопросы

- memcached как способ межпроцессного/межъязыкового взаимодействия;
- Кэширование memcached («кэширование кэша»): в теле процесса, в локальном кэше (eAccelerator и т.п.)
- Другая семантика: memcachedb, memcacheq, и т.п.

Всё!

- Вопросы?
- Контакты:
  - smira@netstream.ru
  - http://smira.ru/