Особенности высоконагруженных WEB систем

Система начинается с данных

Не секрет что разработка практически любой автоматизированной системы начинается с определения формата входных и выходных данных. Данные могут существенно отличаться по своей структуре и организации. Одни могут иметь множественные связи, другие представлять собой просто массив простых типов данных.

Нас в первую очередь интересуют два подхода к хранению и работе с данными SQL и NoSQL.

SQL (Structured Query Language) — язык структурированных запросов, применяемый для создания, модификации и управления данными в реляционных базах данных, основанных на реляционной модели данных. Думаю, подробно останавливаться на рассмотрении одноименного подхода не стоит, так как это первое с чем сталкивается любой, при изучении баз данных.

NoSQL (not only SQL, не только SQL) — ряд подходов, направленных на реализацию моделей баз данных, имеющих существенные отличия от средств языка SQL, характерного для традиционных реляционных баз данных.

Термин ***NoSQL*** был придуман Эриком Эвансом, когда Джоан Оскарсон из Last.fm хотел организовать мероприятие для обсуждения распределенных баз данных с открытым исходным кодом.

Концепция NoSQL не является полным отрицанием языка SQL и реляционной модели. NoSQL — это важный и полезный, но не универсальный инструмент. Одна из проблем классических реляционных БД — это сложности при работе с данными очень большого объема и в высоконагруженных системах. Основная цель NoSQL — расширить возможности БД там, где SQL недостаточно гибок, не обеспечивает должной производительности, и не вытеснять его там, где он отвечает требованиям той или иной задачи.

В июле 2011 компания Couchbase, разработчик CouchDB, Memcached и Membase, анонсировала создание нового SQL-подобного языка запросов — UnQL (Unstructured Data Query Language). Работы по созданию нового языка выполнили создатель SQLite Ричард Гипп (Richard Hipp) и основатель проекта CouchDB Дэмиен Кац (Damien Katz). Разработка передана сообществу на правах общественного достояния

Использование подхода NoSQL пригодятся нам для хранения огромных массивов простой неструктурированной информации, которая не требует связи с другими данными. Примером такой информации может служить многомиллионный список файлов кэшей или изображений. При этом, мы получим значительный выигрыш в производительности по сравнению с реляционным подходом.

NoSQL системы

NoSQL СУБД

Определимся с понятиями.

Масштабируемость — автоматическое распределение данных между несколькими серверами. Такие системы мы называем распределенные базы данных. В них входят Cassandra, HBase, Riak, Scalaris и Voldemort. Если вы используете объем данных, который не может быть обработан на одной машине или если вы не хотите управлять распределением вручную, то это то, что вам нужно.

Следует обратить внимание на следующие вещи: поддержка нескольких датацентров и возможность добавления новых машин в работающий кластер прозрачно для ваших приложений.

Прозрачное добавление машины в кластер

Поддержка нескольких датацентров

Cassandra

X

X

HBase

X

Riak

X

X

Scalaris

X

Voldemort

Необходимо доработать напильником

К нераспределенным базам данных относятся: *CouchDB, MongoDB, Neo4j, Redis и Tokyo Cabinet*. Эти системы можно использовать в качестве «прослойки» для распределенных систем.

Модель данных и запросов

Существует огромное количество моделей данных и API запросов в NoSQL базах данных.

Модель данных

API запросов

Cassandra

Семейство столбцов

Thrift

CoutchDB

Документы

Map / Reduce

HBase

Семейство столбцов

Thrift, REST

MongoDB

Документы

Cursor

Neo4j

Графы

Graph

Redis

Коллекции

Collection

Riak

Документы

Nested hashes, REST

Scalaris

Ключ / Значение

Get / Put

Tokyo Cabinet

Ключ / Значение

Get / Put

Voldemort

Ключ / Значение

Get / Put

Система семейства столбцов (column family) используется в Cassandra и HBase. В обеих системах, у вас есть строки и столбцы, но количество строк не велико: каждая строка имеет переменное число столбцов и столбцы не должны быть определены заранее.

Система ключ/значения простая, и не сложна в реализации, но не эффективна, если вы заинтересованы в запросе или обновлении только части данных. Также трудно реализовать сложные структуры поверх распределенных систем этого типа.

Документно-ориентированные базы данных — это по существу следующий уровень систем ключ/значение, позволяющие связывать вложенные данные с ключом. Поддержка таких запросов более эффективна, чем просто возвращение всего значения.

Neo4J обладает уникальной моделью данных, которая описывает объекты в виде узлов и ребер графа. Для запросов, которые соответствуют этой модели (например, иерархических данных) производительность может оказаться выше на несколько порядков, чем для альтернативных вариантов.

Scalaris уникальна в части использования распределенных транзакций между несколькими ключами.

Система хранения данных

Это вид, в котором данные представлены в системе.

Вид данных

Cassandra

Memtable / SSTable

CoutchDB

Append-only B-Tree

HBase

Memtable / SSTable on HDFS

MongoDB

B-Tree

Neo4j

On-disk linked list

Redis

In-memory with background snapshots

Riak

Hash

Scalaris

In-memory only

Tokyo Cabinet

Hash or B-Tree

Voldemort

Pluggable (primarily BDB MySQL)

Система хранения данных может помочь при оценке нагрузок.

Системы, хранящие данные в памяти, очень-очень быстрые, но не могут работать с данными превышающими размер доступной оперативной памяти. Сохранность таких данных при сбое или отключении питания может стать проблемой. Количество данных

которые могут ожидать записи на может быть очень велико. Некоторые системы, например Scalaris, решают данную проблему с помощью репликации, но Scalaris не поддерживает масштабирование на несколько датацентров.

Memtables / SSTables буферизируют запросы на запись в памяти (Memtable), а после записи добавляют в лог. После накопления достаточного количества записей, Memtable сортируется и записывается на диск, как SSTable. Это позволяет добиться производительности близкой к производительности оперативной памяти, в тоже время избавиться от проблем, актуальных при хранении только в памяти.

B-деревья используются в базах данных уже очень давно. Они обеспечивают надежную поддержку индексирования, однако производительность, при использовании на машинах с магнитными жесткими дисками, очень низкая.
Интересным является использование в CouchDB B-деревьев, только с функцией добавления (append-only B-Trees), что позволяет получить хорошую производительность при записи данных на диск. Достигается это тем, что бинарное дерево не нужно перестраивать при добавлении элемента.

Отдельного рассмотрения заслуживает проект **Memcached** , ставший прародителем для множества других систем.

Memcached — программное обеспечение для кэширования данных в оперативной памяти сервера на основе парадигмы хеш-таблицы. Это высокопроизводительная распределенная система кэширования объектов в оперативной памяти, предназначенная для высоконагруженных интернет-систем.

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, позволяющая хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

С помощью клиентской библиотеки **Memcached** позволяет кэшировать данные в оперативной памяти из множества доступных серверов. Распределение реализуется путем сегментирования данных по значению хэша ключа по аналогии с сокетами

хэш-таблицы. Клиентская библиотека, используя ключ данных, вычисляет хэш и использует его для выбора соответствующего сервера. Ситуация сбоя сервера трактуется как промах кэша, что позволяет повышать отказоустойчивость комплекса за счет наращивания количества memcached серверов и возможности производить их горячую замену.

Memcached имеет множество модификаций, развиваемых в рамках нескольких проектов: *Mycached, membase, Memcache, MemcacheQ, Memcacheddb* и *libMemcached* .

Центральный проект, локомотив концепции NoSQL — Memcached. Один из самых существенных минусов Memcached состоит в том, что сам кэш — весьма ненадежное место хранения данных. Устранить этот недостаток и призваны дополнительные решения: Memcacheddb и membase. На уровне интерфейса (API) эти продукты полностью совместимы с Memcached. Но здесь, при устаревании данных, они сбрасываются на диск (стратегия « db checkpoint »). В таком виде они представляют собой яркий пример «нереляционных баз данных» — персистентных (долговременных) систем распределённого хранения данных в виде пар «ключ-значение».

Следующий продукт, на основе Memcached — MemcacheQ. Это система очередей сообщений, в которой используется очень упрощенный API от Memcached. MemcacheQ образует именованный стек, куда можно записать свои сообщения, а сами данные физически хранятся в БД BerkeleyDB (аналогично Memcacheddb), следовательно, обеспечиваются сохранность, возможность реплицирования и прочее.

LibMemcached — это известная клиентская библиотека, написанная на C++, для работы с уже стандартным протоколом Memcached.

Все нереляционные хранилища, выполненные в виде распределенной системы и хранящие пары «ключ-значение», можно подразделить на два типа: устойчивые и неустойчивые. **Устойчивые** (например, *MemcachedB, membase, Hypertable*) — записывают данные на диск, обеспечивая их сохранность в случае сбоя.
Неустойчивые
(классический Memcached) — хранят ключи в энергозависимой памяти и не гарантируют их сохранность. Неустойчивые хранилища оправдано использовать для кеширования и снижения нагрузки на устойчивые — в этом их неразрывная связь и главное преимущество.

Устойчивые хранилища – это уже полноценные NoSQL базы данных, которые совмещают в себе скорость работы Memcached и позволяют хранить более сложные данные.

Схема frontend+backend

Самая распространенная схема, при которой в роли frontend выступает быстрый и легкий web сервер (например Nginx), а в качестве backend работает Apache.

Давайте рассмотрим преимущества такой схемы на примере. Представим, что web серверу Apache необходимо обслужить порядка 1000 запросов одновременно, причем многие из этих клиентов подключены к интернету по медленным каналам связи. В случае использования Apache мы получим 1000 процессов httpd, на каждый из которых будет выделена оперативная память, и эта память будет занята до тех пор, пока клиент не получит запрошенный контент или не возникнет ошибка.

В случае применения схемы frontend+backend, после того как пришел запрос клиента, Nginx передает запрос Apache и быстро получает ответ. А в случае со статическим контентом (html, картинки, файлы кеша и пр.) Nginx самостоятельно сформирует ответ, не потревожив Apache. Если нам все-таки нужно выполнить логику (php-скрипт), Apache после того как сделал это и отдал ответ Nginx освобождает память, далее с клиентом взаимодействует web сервер Nginx, который как раз и предназначен для раздачи статического контента, большому количеству клиентов, при незначительном потреблении системных ресурсов. В купе с грамотным кэшированием, получаем колоссальную экономию ресурсов сервера и систему, которую по праву можно назвать высоконагруженной.

Рабочие лошадки

Apache - оптимизация производительности

Для схемы frontend+backend производительность Apache не стоит столь остро, но если Вам дорога каждая микросекунда процессорного времени и каждый байт оперативной памяти, то следует уделить внимание этому вопросу.

Самый «крутой» способ – увеличить производительность сервера – поставить более шустрый процессор(ы) и побольше памяти, но мы с Вами пойдем по менее радикальному

пути для начала. Ускорим работу Apache путем оптимизации его конфигурации. Существуют конфигурации, которые можно применить только при пересборке Apache, другие же можно применять без перекомпиляции сервера.

Загружайте только необходимые модули

Большая часть функций Apache реализуется при помощи модулей. При этом эти модули могут быть как «вшиты» в ту или иную сборку, так и загружаться в виде динамических библиотек (DSO). Большинство современных дистрибутивов поставляет Apache с набором DSO, так что не нужные модули можно отключить без перекомпиляции.

Уменьшив количество модулей, Вы уменьшите объем потребляемой памяти. Если вы решили скомпилировать Apache самостоятельно, то либо тщательно подходите к выбору списка модулей, которые Вы хотите включить, либо скомпилируйте их как DSO, используя apxs в Apache1 и apxs2 в Apache2. Чтобы отключить ненужные DSO-модули, достаточно закомментировать лишние строчки LoadModule в httpd.conf. Если скомпилировать модули статически, Apache будет потреблять чуть меньше памяти, но Вам придется каждый раз его перекомпилировать, чтобы отключить или включить тот или иной модуль.

Выбирайте подходящий MPM

Для обработки запроса в Apache выделяется свой процессе или поток. Эти процессы работают в соответствии с одной из MPM (Мультипроцессорная модель). Выбор MPM зависит от многих факторов, таких как наличие поддержки потоков в ОС, объема свободной памяти, а также требований стабильности и безопасности.

Если безопасность превыше производительности, выбирайте peruser или Apache-itk. Если важнее производительность, обратите внимание на prefork или worker.

Название

Разработчик

Поддерживаемые OS

Описание

Назначение

Статус

worker

Apache Software Foundation

Linux, FreeBSD

Гибридная мультипроцессорно-мультипоточная модель. Сохраняя стабильность мультипроцесс

Среднезагруженные веб-серверы.

Стабильный.

pre-fork

Apache Software Foundation

Linux, FreeBSD

MPM, основанная на предварительном создании отдельных процессов, не использующая механи

Большая безопасность и стабильность за счёт изоляции процессов друг от друга, сохранение со

Стабильный.

perchild

Apache Software Foundation

Linux

Гибридная модель, с фиксированным количеством процессов.

Высоконагруженные серверы, возможность запуска дочерних процессов используя другое имя п

В разработке, нестабильный.

netware

Apache Software Foundation

Novell NetWare

Мультипоточная модель, оптимизированная для работы в среде NetWare.

Серверы Novell NetWare

Стабильный.

winnt

Apache Software Foundation

Microsoft Windows

Мультипоточная модель, созданная для операционной системы Microsoft Windows.

Серверы под управлением Windows Server.

Стабильный.

Apache-ITK

Steinar H. Gunderson

Linux, FreeBSD

MPM, основанная на модели prefork. Позволяет запуск каждого виртуального хоста под отдельн

Хостинговые серверы, серверы, критичные к изоляции пользователей и учёту ресурсов.

Стабильный.

peruser

Sean Gabriel Heacock

Linux, FreeBSD

Модель, созданная на базе MPM perchild. Позволяет запуск каждого виртуального хоста под отд

Обеспечение повышенной безопасности, работа с библиотеками, не поддерживающими threads.

Стабильная версия от 4 октября 2007 года, экспериментальная — от 10 сентября 2009 года.

Для смены MPM требуется перекомпиляция Apache. Для этого удобнее взять source-based дистрибутив.

DNS lookup

Директива HostnameLookups включает обратные DNS запросы, при этом в логи пишутся dns-хосты клиентов вместо ip-адресов. Это существенно замедляет обработку запроса, т.к. запрос не обработается, пока не будет получен ответ от DNS-сервера. Следите, чтобы эта директива всегда была выключена (HostnameLookups Off). Если необходимы dns-адреса, можно «прогнать» лог в утилите logresolve.

Кроме того, следите, чтобы в директивах Allow from и Deny From использовались ip-адреса а не доменные имена. В противном случае Apache будет делать два dns запроса (обратный и прямой), чтобы узнать ip и убедиться что клиент валиден.

AllowOverride

Если директива AllowOverride не установлена в None, Apache попытается открыть .htaccess файлы в каждой директории, которую он посещает и во всех директориях выше нее. Например:

DocumentRoot /var/www/html

<Directory /var/www/html/>

AllowOverride all

</Directory>

Если будет запрошен /index.html, Apache попытается открыть (и интерпретировать) файлы /.htaccess, /var/.htaccess, /var/www/.htaccess, и /var/www/html/.htaccess. Очевидно, что это увеличивает время обработки запроса. Так что, если вам нужен .htaccess только для одной директории, разрешите его только для нее:

DocumentRoot /var/www/html

<Directory />

AllowOverride None

</Directory>

<Directory /var/www/html/>

AllowOverride all

</Directory>

FollowSymLinks и SymLinksIfOwnerMatch

Если для каталога включена опция FollowSymLinks, Apache будет следовать по символическим ссылкам в этом каталоге . Если включена опция SymLinksIfOwnerMatch, Apache будет следовать по символическим ссылкам, только если владелец файла или каталога на которую указывает эта ссылка, совпадает с владельцем указанного каталога. Поэтому при включенной опции SymLinksIfOwnerMatch Apache делает больше системных запросов. Кроме того, дополнительные системные запросы требуются, когда

FollowSymlinks не определен. Следовательно, наиболее оптимальным для производительности будет включение опции FollowSymlinks, конечно, если политика безопасности позволяет это сделать.

Content Negotiatio

Механизм, определенный в HTTP спецификации, который позволяет обслуживать различные версии документа (представления ресурса), для одного и того же URI, чтобы клиент смог определить, какая версия лучше соответствует его возможностям. Когда клиент отправляет запрос на сервер, то сообщает серверу, какие типы файлов он понимает. Каждому типу соответствует рейтинг, описывающий насколько хорошо клиент его понимает. Таким образом, сервер, способен предоставить версию ресурса, который наилучшим образом соответствует потребностям клиента.

Нетрудно понять, чем это грозит для производительности сервера, поэтому избегайте применения content negotiaion.

MaxClients

Директива MaxClients устанавливает максимальное количество параллельных запросов, которые будет поддерживать сервер. Значение MaxClient не должно быть слишком маленьким, иначе многим клиентам будет отказано. Нельзя устанавливать слишком большое количество – это грозит нехваткой ресурсов и «падением» сервера. Ориентировочно, MaxClients = количество памяти выделенное под веб-сервер / максимальный размер порожденного процесса или потока. Для статических файлов Apache использует около 2-3 Мб на процесс, для динамики (php, cgi) – зависит от скрипта, но обычно около 16-32 Мб. Если сервер уже обслуживает MaxClients запросов, новые запросы попадают в очередь, размер которой устанавливается директивой ListenBacklog.

MinSpareServers, MaxSpareServers, и StartServers

Создание потока, и особенно процесса – ресурсоемкая операция, поэтому Apache создает их про запас. Директивы MaxSpareServers и MinSpareServers устанавливают минимальное и максимальное число процессов/потоков, которые должны быть готовы принять запрос. Если значение MinSpareServers слишком мало и пришло много запросов, Apache начнет создавать много новых процессов/потоков, что создаст лишнюю нагрузку в пиковые моменты. Если MaxSpareServers слишком велико, Apache будет излишне нагружать систему, даже если число запросов минимально.

Опытным путем нужно попдобрать такие значения, чтобы Apache не создавал более 4 процессов/потоков в секунду. Если он создаст более 4, в ErrorLog будет сделана соответствующая запись – сигнал того что MinSpareServers нужно увеличить.

MaxRequestsPerChild

Директива MaxRequestsPerChild определяет максимальное число запросов, которое может обработать один дочерний процесс/поток прежде чем он завершиться. По умолчанию значение установлено в 0, что означает, что не будет завершен никогда. Рекомендуется установить MaxRequestsPerChild равное числу запросов за час. Это не создаст лишней нагрузки на сервер и, в то же время, поможет избавиться от проблем с утечкой памяти в дочерних процессах (например, если вы используете нестабильную версию php).

KeepAlive и KeepAliveTimeout

KeepAlive позволяет делать несколько запросов в одном TCP-подключении. При использовании схемы frontend+backend, эти директивы не актуальны.

HTTP-сжатие

Сейчас все современные клиенты и практически все сервера поддерживают HTTP-сжатие. Использование сжатия позволяет понизить трафик между клиентом и сервером до 4-х рах, повышая при этом нагрузку на процессор сервера. Но, если сервер посещает много клиентов с медленными каналами, сжатие способно снизить нагрузку посредством уменьшения времени передачи сжатого ответа. При этом ресурсы, занятые дочерним процессом освобождаются быстрее, и уменьшается число одновременных запросов. Это особенно заметно в условиях ограничения памяти.

Отмечу, что не следует устанавливать степень сжатия gzip более 5, так как существенно возрастает нагрузка на процессор, а степень сжатия растет незначительно. Также, не следует сжимать файлы, формат которых уже подразумевает сжатие – это практически все мультимедийные файлы и архивы.

Кеширование на стороне клиента

Не забывайте устанавливать Expires заголовки для статических файлов (модуль

mod_expires). Если файл не изменяется, то всегда следует дать указание клиенту закэшировать его. При этом у клиента будут быстрее загружаться страницы, а сервер избавиться от лишних запросов.

Отключение логов

Отключение логов помогает временно справиться с нагрузкой в пиковые моменты. Эта мера существенно сокращает нагрузку на все виды ПО и является универсальным средством в критической ситуации. Естественно, при очевидных недостатках, не может рекомендоваться к применению и служит лишь временным решением проблемы.

Nginx

Простой и легкий веб-сервер, специально предназначенный для обработки статических запросов. Причина его производительности в том, что рабочие процессы обслуживают одновременно множество соединений, мультиплексируя их вызовами операционной системы select, epoll (Linux) и kqueue (FreeBSD). Сервер имеет эффективную систему управления памятью с применением пулов. Ответ клиенту формируется в буферах, которые хранят данные либо в памяти, либо указывают на отрезок файла. Буферы объединяются в цепочки, определяющие последовательность, в которой данные будут переданы клиенту. Если операционная система поддерживает эффективные операции ввода-вывода, такие как **writev** и **sendfile** , то **Nginx,** по возможности, применяет их.

При использовании в связке с Apache, Nginx настраивается на обработку статики и используется для балансировки нагрузки. Подавляющее время занимается лишь тем, что отдает статический контент, делает это очень быстро и с минимальными накладными расходами.

Lighttpd

«Веб-сервер, разрабатываемый с расчётом на быстроту и защищённость, а также соответствие стандартам.» – википедия

Является альтернативой Nginx и применяется для тех же целей.

Акселераторы PHP

Принцип работы таких продуктов в том, что они кэшируют байт-код скрипта и позволяют снизить нагрузку на интерпретатор PHP.

Существующие решения

The Alternative PHP Cache —был задуман, как бесплатный, открытый и стабильный фреймворк для кэширования и оптимизации исходного кода PHP. Поддерживает PHP4 и PHP5, включая 5.3.

eAccelerator — это свободный открытый проект, выполняющий также роли акселератора, оптимизатора и распаковщика. Имеет встроенные функции динамического кэширования контента. Имеется возможность оптимизации PHP-скриптов. Поддерживает PHP4 и PHP5, включая 5.3.

PhpExpress бесплатный ускоритель обработки PHP скриптов на веб-сервере. Также обеспечивает поддержку загрузки файлов закодированных через Nu-Coder. Поддерживает PHP4 и PHP5, включая 5.3

XCache поддерживает PHP4 и PHP5, включая 5.3. Начиная с версии 2.0.0 (release candidate от 2012-04-05) включена поддержка PHP 5.4.

Windows Cache Extension for PHP - PHP-акселератор для Microsoft IIS (BSD License). Поддерживает только PHP (5.2 и 5.3).

Логика кэширования

«Кэшировать, кэшировать и еще раз кэшировать!» - вот девиз высоконагруженной системы.

Давайте представим себе идеальный высоконагруженный сайт. Сервер получает http запрос от клиент. Frontend сопоставляет запрос с физическим файлом на сервере и, если тот существует, отдает его. Загрузку скриптов и картинок опустим, так как это в большинстве своем статика и отдается по такому же принципу. Далее, если физически

файл не существует, frontend обращается с этим запросом к backend-у, который занимается обработкой логики (скриптов php и т.д.). Backend должен решить кэшировать ли данный запрос и создать файл в определенном месте, который и будет отдаваться frontend-ом в дальнейшем. Таким образом, мы навсегда закэшировали данный запрос и сервер будет обрабатывать его максимально быстро с минимально возможной нагрузкой на сервер.

Данный идеальный пример подходит для страниц, содержание которых не меняется с течением времени, либо меняется редко. На практике же мы имеем страницы, содержимое которых может меняться с каждым последующим запросом. Вернее часть этого содержимого. Примером такого содержимого может служить пользовательская информация, которая должна меняться с незаметной для пользователя задержкой или отображаться в реальном времени (обновляться при каждой перезагрузке страницы). Тут перед нами возникает задача, которая сводится к разделению динамических и статических данных на странице.

Самым удобным и распространенным способом разделения данных является разделение страницы на блоки. Это логично и удобно, ведь страница, с точки зрения верстки, состоит из блоков. Избежать логики в этом случае естественно не получится, но логика эта будет обрабатываться с наименьшими затратами.

Таким образом, запрос клиента (кроме запроса статики) переадресуется на backend и его обработка сводится к следующим действиям:

1. Получение информации о блоках, которые будут на странице.
2. Проверка информации о кэшах для каждого блока. Кэш может не существовать или нуждаться в обновлении. В этом случае генерируем файл кэша. Если блок не должен кэшироваться выполняем соответствующую логику. Информацию о кэшах можно хранить в nosql базе данных или в файловой структуре. Тут требование одно: получение этой информации должно занимать минимум ресурсов.
3. Формируем html страницы. Закэшированные блоки встраиваем при помощи ssi инструкции (вставляется ссылка на файл кэша), что позволит существенно экономить память.
4. Страница попадает на frontend, который производит замену всех ssi инструкций на содержимое файлов и отдает страницу клиенту.

Также, распространенным является кэширование результатов выполнения функции или

метода класса. При этом, мы передаем кэширующей функции ссылку на объект (если вызываем метод), имя метода или функции (если это глобальная функция) и параметры, предназначающиеся этому методу или функции. Кэширующая функция проверит наличие файла кэша, при необходимости, сформирует его или прочтет, а затем вернет результат.

Это общее описания принципа работы высоконагруженного сайта. Конкретная реализация будет отличаться деталями, но концепция останется прежней.

Картинки, пикчи, тумбочки

Оказывается изображение тоже можно кэшировать. Зачем? Спросите Вы. В принципе, после загрузки на сервер у нас уже есть файл, который frontend быстренько выплюнет при необходимости. Но часто нам требуется получить на основе уже имеющейся картинки другое изображение (например, других размеров). Допустим, нам нужна миниатюра изображения – thumbnail (тумбочка жарг.). В этом случае, нам достаточно сформировать путь к бедующему файлу уменьшенной картинки и отдать страницу клиенту.

1. Клиент, получив исходный код страницы, начинает подгружать статику и обращается с запросом на несуществующую пока картинку к frontend-у.
2. Frontend переадресует запросы к несуществующим изображениям на backend.
3. Backend анализирует запрос, формирует файл изображения и отдает бинарные данные с соответствующим http-заголовком.
4. Все последующие запросы будут отдаваться frontend-ом.