Wednesday, April 26, 2017

System Design - Cache

https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#COHDG5177

Key value cache

KV cache is like a giant hash map and used to reduce the latency of data access, typically by

1. Putting data from slow and cheap media to fast and expensive ones.
2. Indexing from tree-based data structures of `O(log n)` to hash-based ones of `O(1)` to read and write

There are various cache policies like read-through/write-through(or write-back), and cache-aside. By and large, Internet services have a read to write ratio of 100:1 to 1000:1, so we usually optimize for read.

In distributed systems, we choose those policies according to the business requirements and contexts, under the guidance of CAP theorem.

Regular Patterns

- Read
  - Read-through: the clients read data from the database via the cache layer. The cache returns when the read hits the cache; otherwise, it fetches data from the database, caches it, and then return the vale.
- Write
  - Write-through: clients write to the cache and the cache updates the database. The cache returns when it finishes the database write.
  - Write-behind / write-back: clients write to the cache, and the cache returns immediately. Behind the cache write, the cache asynchronously writes to the database.
  - Write-around: clients write to the database directly, around the cache.

Cache-aside pattern

When a cache does not support native read-through and write-through operations, and the resource demand is unpredictable, we use this cache-aside pattern.

- Read: try to hit the cache. If not hit, read from the database and then update the cache.
- Write: write to the database first and then delete the cache entry. A common pitfall here is that people mistakenly update the cache with the value, and double writes in a high concurrency environment will make the cache dirty.

There are still chances for dirty cache in this pattern. It happens when these two cases are met in a racing condition:

1. read database and update cache
2. update database and delete cache

Where to put the cache?

- client-side
- distinct layer
- server-side

What if data volume reaches the cache capacity? Use cache replacement policies

- LRU(Least Recently Used): evict the most recently used entries and keep the most recently used ones.
- LFU(Least Frequently Used): evict the most frequently used entries and keep the most frequently used ones.
- ARC(Adaptive replacement cache): it has a better performance than LRU. It is achieved by keeping both the most frequently and frequently used entries, as well as a history for eviction. (Keeping MRU+MFU+eviction history.)

Who are the King of the cache usage?

Facebook TAO
https://lethain.com/introduction-to-architecting-systems-for-scale/#caching

Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.

Caching consists of: precalculating results (e.g. the number of visits from each referring domain for the previous day), pre-generating expensive indexes (e.g. suggested stories based on a user's click history), and storing copies of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL.

Popular Posts

Archives
thought-works: Object Oriented design for Elevator in a multi-storied apartment
Design a chess game using OO principles | Runhe Tian Coding Practice
How to design a tiny URL or URL shortener?
[CC150v5] 8.4 Design a Parking Lot - Shuatiblog.com
Google Map Architecture
Google Calendar Architecture
Design a chat server | Hello World
System Design Interview Summary
System Design Interview

Labels

- Review (572)
- System Design (334)
- System Design - Review (198)
- Java (189)
- Coding (75)
- Interview-System Design (65)
- Interview (63)
- Book Notes (59)
- Coding - Review (59)
- to-do (45)
- Linux (41)
- Knowledge (39)
- Interview-Java (35)
- Knowledge - Review (32)
- Database (31)
- Design Patterns (31)
- Big Data (29)
- Product Architecture (28)
- MultiThread (27)
- Soft Skills (27)
- Concurrency (26)
- Cracking Code Interview (26)
- Miscs (25)
- Distributed (24)
- OOD Design (24)
- Career (22)
- Google (21)
- Interview - Review (21)
- Java - Code (21)
- Operating System (21)
- Interview Q&A (20)
- System Design - Practice (20)
- Tips (19)
- Algorithm (17)
- Company - Facebook (17)
- Security (17)
- How to Ace Interview (16)
- Brain Teaser (14)
- Linux - Shell (14)
- Redis (14)
- Tools (14)
- Code Quality (13)
- Search (13)
- Spark (13)
- Spring (13)

Search This Blog

Search

Blog Archive

- 2021 (3)
- 2019 (111)
- 2018 (116)
- 2017 (65)
  - December (4)
  - November (3)
  - October (5)
  - September (4)
  - August (9)
  - July (10)
  - June (3)
  - May (4)
  - April (12)
    - Soft Skills for Tech interviews
    - System Design - Asynchronism
    - System Design - Cache
    - System Design - Database Misc
    - Design Post System
    - Cardinality Estimation
    - System Design Interview Misc
    - Stories about Scalability
    - Redis Misc
    - Java Scheduler
    - Retrospectives
    - How to make software design decisions
  - March (4)
  - February (1)
  - January (6)
- 2016 (342)
- 2015 (724)
- 2014 (112)
- 2013 (2)
- 2011 (8)

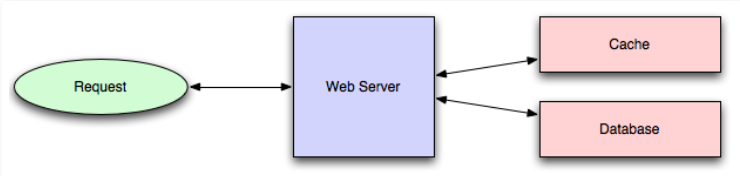
Popular Posts

Archives
Design a chess game using OO principles | Runhe Tian Coding Practice
thought-works: Object Oriented design for Elevator in a multi-storied apartment
How to design a tiny URL or URL shortener?
Google Map Architecture
[CC150v5] 8.4 Design a Parking Lot - Shuatiblog.com
Google Calendar Architecture
Design a chat server | Hello World
Design a news feed system
Design Hit Counter - how to count number of requests in last second, minute and hour - Stack Overflow

In practice, caching is important earlier in the development process than load-balancing, and starting with a consistent caching strategy will save you time later on. It also ensures you don't optimize access patterns which can't be replicated with your caching mechanism or access patterns where performance becomes unimportant after the addition of caching (I've found that many heavily optimized Cassandra applications are a challenge to cleanly add caching to if/when the database's caching strategy can't be applied to your access patterns, as the datamodel is generally inconsistent between the Cassandra and your cache).

Application vs. database caching

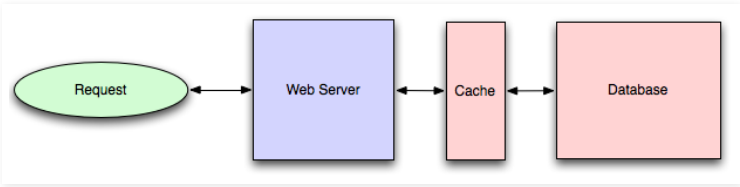
There are two primary approaches to caching: application caching and database caching (most systems rely heavily on both).



Application caching requires explicit integration in the application code itself. Usually it will check if a value is in the cache; if not, retrieve the value from the database; then write that value into the cache (this value is especially common if you are using a cache which observes the least recently used caching algorithm). The code typically looks like (specifically this is a *read-through cache*, as it reads the value from the database into the cache if it is missing from the cache):

```
key = "user.%s" % user_id
user_blob = memcache.get(key)
if user_blob is None:
    user = mysql.query("SELECT * FROM users WHERE user_id=\"%s\"", user_id)
    if user:
        memcache.set(key, json.dumps(user))
    return user
else:
    return json.loads(user_blob)
```

The other side of the coin is database caching.



When you flip your database on, you're going to get some level of default configuration which will provide some degree of caching and performance. Those initial settings will be optimized for a generic usecase, and by tweaking them to your system's access patterns you can generally squeeze a great deal of performance improvement.

The beauty of database caching is that your application code gets faster "for free", and a talented DBA or operational engineer can uncover quite a bit of performance without your code changing a whit (my colleague Rob Coli spent some time recently optimizing our configuration for Cassandra row caches, and was succcessful to the extent that he spent a week harassing us with graphs showing the I/O load dropping dramatically and request latencies improving substantially as well).

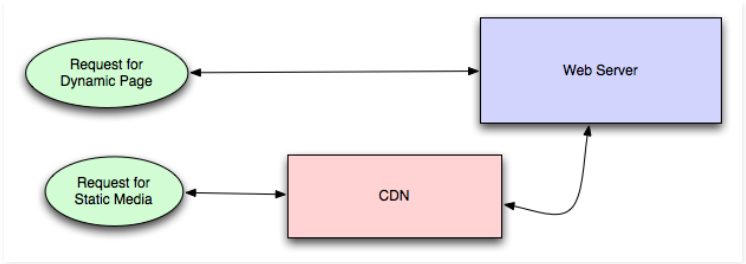
In-memory caches

The most potent–in terms of raw performance–caches you'll encounter are those which store their entire set of data in memory. Memcached and Redis are both examples of in-memory caches (caveat: Redis can be configured to store some data to disk). This is because accesses to RAM are orders of magnitude faster than those to disk.

On the other hand, you'll generally have far less RAM available than disk space, so you'll need a strategy for only keeping the hot subset of your data in your memory cache. The most straightforward strategy is least recently used, and is employed by Memcache (and Redis as of 2.2 can be configured to employ it as well). LRU works by evicting less commonly used data in preference of more frequently used data, and is almost always an appropriate caching strategy.

Content distribution networks

A particular kind of cache (some might argue with this usage of the term, but I find it fitting) which comes into play for sites serving large amounts of static media is the *content distribution network.*



CDNs take the burden of serving static media off of your application servers (which are typically optimzed for serving dynamic pages rather than static media), and provide geographic distribution. Overall, your static assets will load more quickly and with less strain on your servers (but a new strain of business expense).

In a typical CDN setup, a request will first ask your CDN for a piece of static media, the CDN will serve that content if it has it locally available (HTTP headers are used for configuring how the CDN caches a given piece of content). If it isn't available, the CDN will query your servers for the file and then cache it locally and serve it to the requesting user (in this configuration they are acting as a read-through cache).

If your site isn't yet large enough to merit its own CDN, you can ease a future transition by serving your static media off a separate subdomain (e.g. `static.example.com`) using a lightweight HTTP server like Nginx, and cutover the DNS from your servers to a CDN at a later date.

Cache invalidation

While caching is fantastic, it does require you to maintain consistency between your caches and the source of truth (i.e. your database), at risk of truly bizarre applicaiton behavior.

Solving this problem is known as *cache invalidation*.

If you're dealing with a single datacenter, it tends to be a straightforward problem, but it's easy to introduce errors if you have multiple codepaths writing to your database and cache (which is almost always going to happen if you don't go into writing the application with a caching strategy already in mind). At a high level, the solution is: each time a value changes, write the new value into the cache (this is called a *write-through* cache) or simply delete the current value from the cache and allow a read-through cache to populate it later (choosing between read and write through caches depends on your application's details, but generally I prefer write-through caches as they reduce likelihood of a stampede on your backend database).

Invalidation becomes meaningfully more challenging for scenarios involving fuzzy queries (e.g if you are trying to add application level caching in-front of a full-text search engine like SOLR), or modifications to unknown number of elements (e.g. deleting all objects created more than a week ago).

In those scenarios you have to consider relying fully on database caching, adding aggressive expirations to the cached data, or reworking your application's logic to avoid the issue (e.g. instead of `DELETE FROM a WHERE...`, retrieve all the items which match the criteria, invalidate the corresponding cache rows and then delete the rows by their primary key explicitly).

https://github.com/FreemanZhang/system-design#solutions

Thundering herd problem

Def

- Many readers read an empty value from the cache and subseqeuntly try to load it from the database. The result is unnecessary database load as all readers simultaneously execute the same query against the database.
- Let's say you have [lots] of webservers all hitting a single memcache key that caches the result of a slow database query, say some sort of stat for the homepage of your site. When the memcache key expires, all the webservers may think "ah, no key, I will calculate the result and save it back to memcache". Now you have [lots] of servers all doing the same expensive DB query.
- Stale date solution: The first client to request data past the stale date is asked to refresh the data, while subsequent requests are given the stale but not-yet-expired data as if it were fresh, with the understanding that it will get refreshed in a 'reasonable' amount of time by that initial request
    - When a cache entry is known to be getting close to expiry, continue to server the cache entry while reloading it before it expires.
    - When a cache entry is based on an underlying data store and the underlying data store changes in such a way that the cache entry should be updated, either trigger an (a) update or (b) invalidation of that entry from the data store.
- Add entropy back into your system: If your system doesn't jitter then you get thundering herds.
    - For example, cache expirations. For a popular video they cache things as best they can. The most popular video they might cache for 24 hours. If everything expires at one time then every machine will calculate the expiration at the same time. This creates a thundering herd.

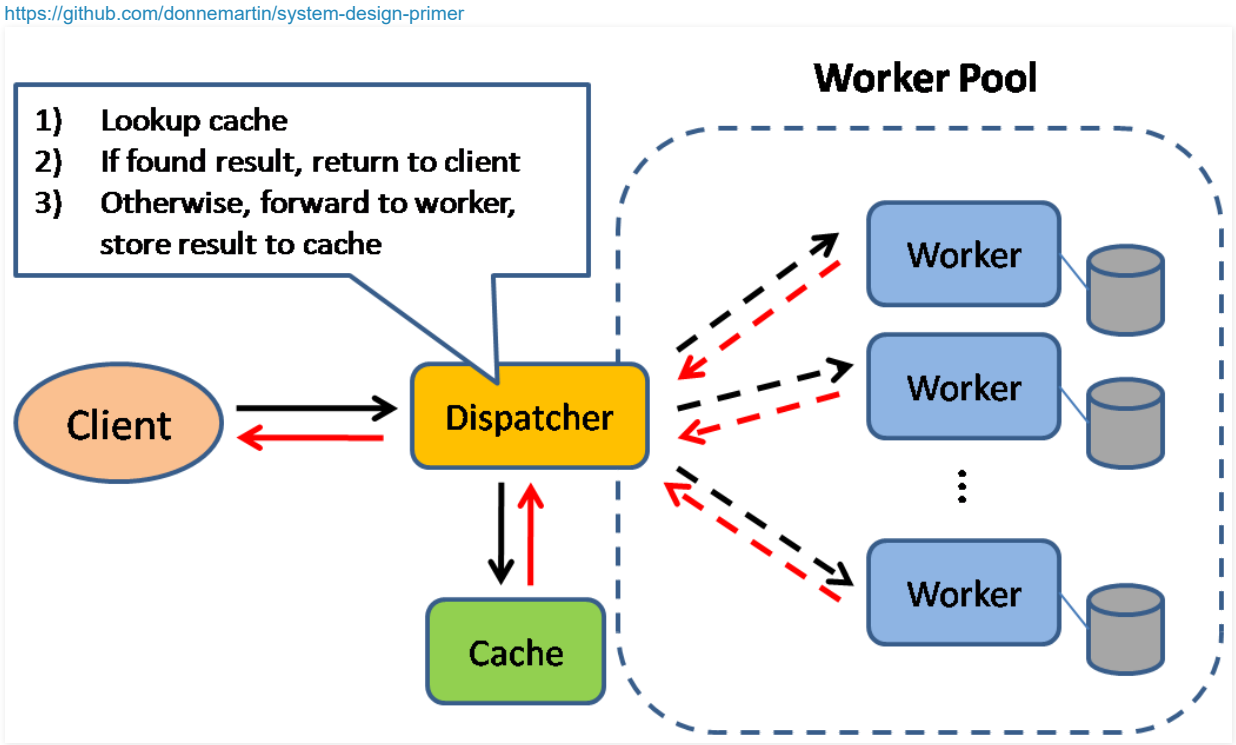
    - By jittering you are saying randomly expire between 18-30 hours. That prevents things from stacking up. They use this all over the place. Systems have a tendency to self synchronize as operations line up and try to destroy themselves. Fascinating to watch. You get slow disk system on one machine and everybody is waiting on a request so all of a sudden all these other requests on all these other machines are completely synchronized. This happens when you have many machines and you have many events. Each one actually removes entropy from the system so you have to add some back in.

- No expire solution: If cache items never expire then there can never be a recalculation storm. Then how do you update the data? Use cron to periodically run the calculation and populate the cache. Take the responsibility for cache maintenance out of the application space. This approach can also be used to pre-warm the the cache so a newly brought up system doesn't peg the database.
    - The problem is the solution doesn't always work. Memcached can still evict your cache item when it starts running out of memory. It uses a LRU (least recently used) policy so your cache item may not be around when a program needs it which means it will have to go without, use a local cache, or recalculate. And if we recalculate we still have the same piling on issues.
    - This approach also doesn't work well for item specific caching. It works for globally calculated items like top N posts, but it doesn't really make sense to periodically cache items for user data when the user isn't even active. I suppose you could keep an active list to get around this limitation though.

Scaling Memcached at Facebook

- In a cluster:
    - Reduce latency
        - Problem: Items are distributed across the memcached servers through consistent hashing. Thus web servers have to rountinely communicate with many memcached servers to satisfy a user request. As a result, all web servers communicate with every memcached server in a short period of time. This all-to-all communication pattern can cause incast congestion or allow a single server to become the bottleneck for many web servers.
        - Solution: Focus on the memcache client.
    - Reduce load
        - Problem: Use memcache to reduce the frequency of fetching data among more expensive paths such as database queries. Web servers fall back to these paths when the desired data is not cached.
        - Solution: Leases; Stale values;
    - Handling failures
        - Problem:
            - A small number of hosts are inaccessible due to a network or server failure.
            - A widespread outage that affects a significant percentage of the servers within the cluster.
        - Solution:
            - Small outages: Automated remediation system.
            - Gutter pool
    - In a region: Replication
    - Across regions: Consistency

https://github.com/donnemartin/system-design-primer



Source: Scalable system design patterns

Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

Client caching

Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

CDN caching

CDNs are considered a type of cache.

Web server caching

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

Database caching

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

Application caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has the following additional features:

- Persistence option
- Built-in data structures such as sorted sets and lists

There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:

- Row level
- Query-level
- Fully-formed serializable objects
- Fully-rendered HTML

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

Caching at the database query level

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries
- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

Caching at the object level

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

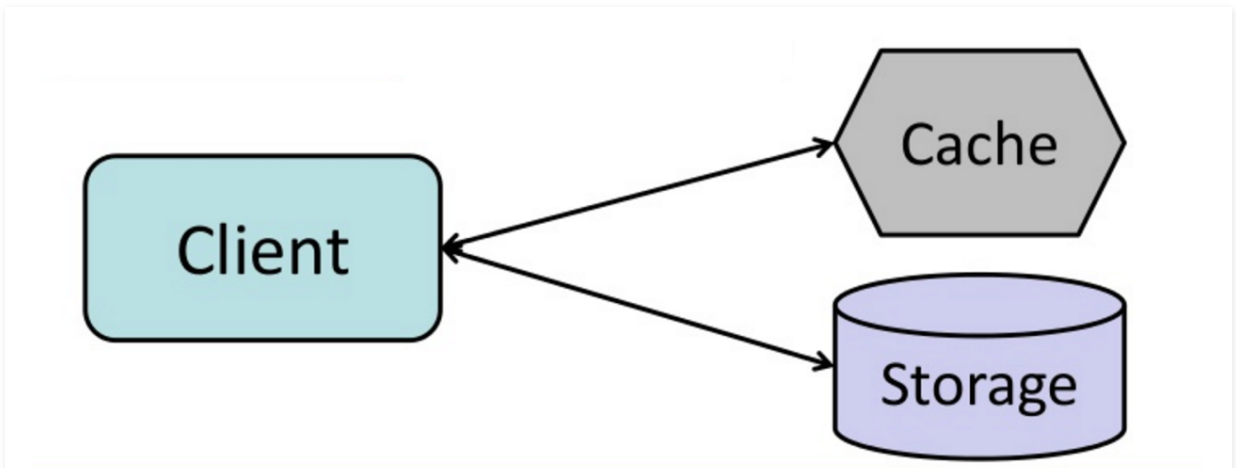
Suggestions of what to cache:

- User sessions
- Fully rendered web pages
- Activity streams
- User graph data

When to update the cache

Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.

Cache-aside



Source: From cache to in-memory data grid

The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
```

```
            cache.set(key, json.dumps(user))
    return user
```

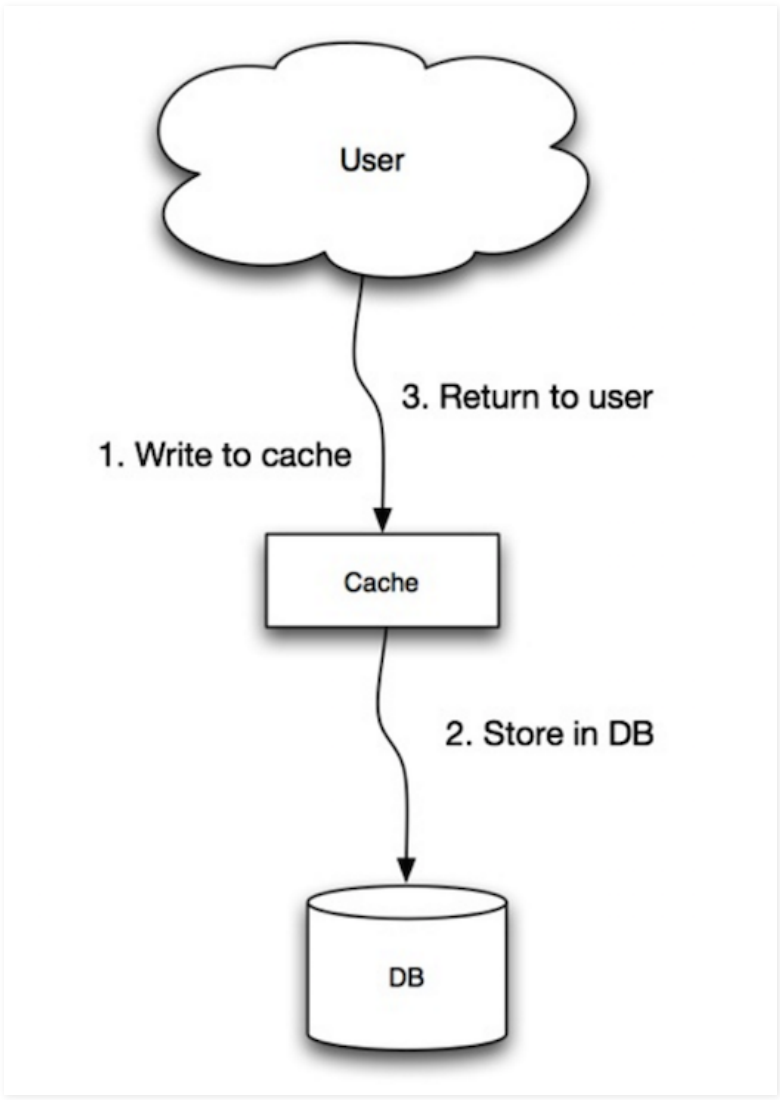
Memcached is generally used in this manner.

Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

Disadvantage(s): cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through



Source: Scalability, availability, stability, patterns

The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return

Application code:

```
set_user(12345, {"foo":"bar"})
```

Cache code:

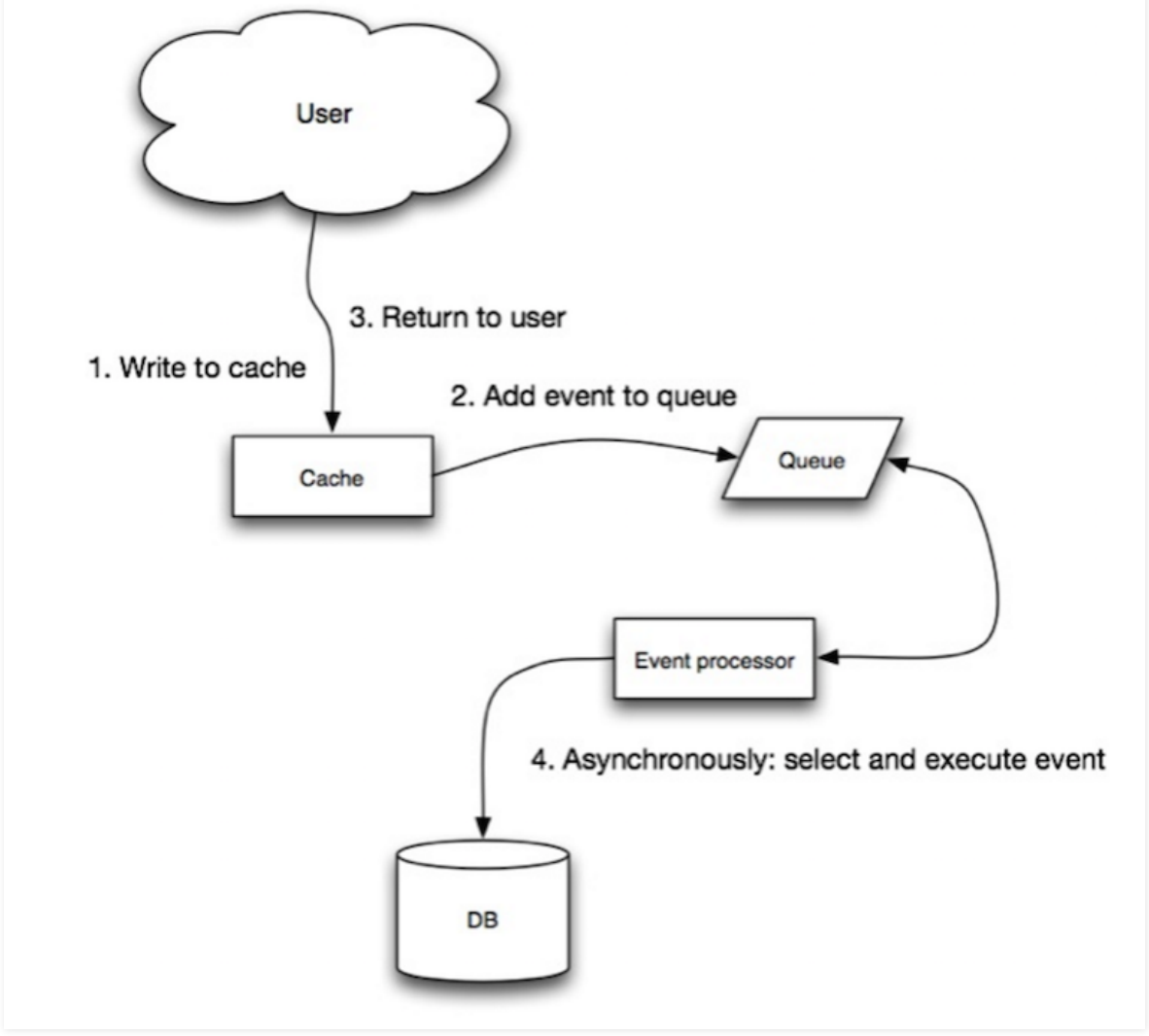
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

Disadvantage(s): write through

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never read, which can be minimized with a TTL.

Write-behind (write-back)



Source: Scalability, availability, stability, patterns

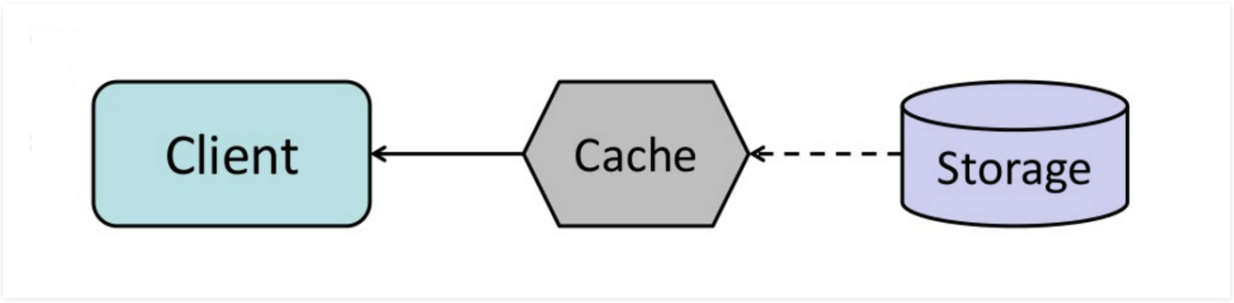
In write-behind, the application does the following:

- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

Disadvantage(s): write-behind

- There could be data loss if the cache goes down prior to its contents hitting the data store.
- It is more complex to implement write-behind than it is to implement cache-aside or write-through.

Refresh-ahead



Source: From cache to in-memory data grid

You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

Disadvantage(s): refresh-ahead

- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

Disadvantage(s): cache

- Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
- Need to make application changes such as adding Redis or memcached.
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- From cache to in-memory data grid
- Scalable system design patterns
- Introduction to architecting systems for scale
- Scalability, availability, stability, patterns
- Scalability
- AWS ElastiCache strategies
- Wikipedia

http://www.ehcache.org/documentation/2.8/recipes/thunderingherd.html

Many readers read an empty value from the cache and subseqeuntly try to load it from the database. The result is unnecessary database load as all readers simultaneously execute the same query against the database.

mplement the cache-as-sor pattern by using a BlockingCache or SelfPopulatingCache included with Ehcache.

Using the BlockingCache Ehcache will automatically block all threads that are simultaneously requesting a particular value and let one and only one thread through to the database. Once that thread has populated the cache, the other threads will be allowed to read the cached value.

Even better, when used in a cluster with Terracotta, Ehcache will automatically coordinate access to the cache across the cluster, and no matter how many application servers are deployed, still only one user request will be serviced by the database on cache misses.

Scaling memcached at Facebook
https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/
https://www.usenix.org/node/172909

Responsiveness is essential for web services. Speed drives user engagement, which drives revenue. To reduce response latency, modern web services are architected to serve as much as possible from in-memory caches. The structure is familiar: a database is split among servers with caches for scaling reads.

Over time, caches tends to accumulate more responsibility in the storage stack. Effective caching makes issuing costly queries feasible, even on the critical response path. Intermediate or highly transient data may never need to be written to disk, assuming caches are sufficiently reliable. Eventually, the state management problem is flipped on its head. The cache becomes the primary data store, without which the service cannot function.
This is a common evolution path with common pitfalls. Consistency, failure handling, replication, and load balancing are all complicated by relying on soft-state caches as the de-facto storage system

The techniques described provide a valuable roadmap for scaling the typical sharded SQL stack: improvements to the software implementation, coordination between the caching layer and the database to manage consistency and replication, as well as failover handling tuned to the operational needs of a billion-user system.

While the ensemble of techniques described by the authors has proven remarkably scalable, the semantics of the system as a whole are difficult to state precisely. Memcached may lose or evict data without notice. Replication and cache invalidation are not transactional, providing no consistency guarantees. Isolation is limited and failurehandling is workload dependent. Won't such an imprecise system be incomprehensible to developers and impossible to operate?
For those who believe that strong semantics are essential for building large-scale services, this paper provides a compelling counter-example. In practice, performance, availability, and simplicity often outweigh the benefits of precise semantics, at leastfor one very large-scale web service.
http://abineshtd.blogspot.com/2013/04/notes-on-scaling-memcache-at-facebook.html
Memcache is used as demand filled look-aside cache where data updates result in deletion of keys in cache

1. Any change must impact user-facing or operational issue. Limited scope optimisations are rarely considered.
2. Tolerant to exposing slightly stale data in exchange for insulating backend store.

A memcache cluster houses many memcached servers across which data is distributed using consistent hashing. In this scenario, it is possible for one web server to communicate with all other memcached servers in a short period of time(incast congestion: see footnote 1 for longer explanation).

http://nil.csail.mit.edu/6.824/2015/notes/l-memcached.txt

FB 系统设计真题解析 & 面试官评分标准
Design a photo reference counting system at fb scale

这是考设计distributed counting system吗？这种题一般是考察vector lock还是什么呢？
学员提问
我感觉是不是就是2种思路，一种是：client向central server传+1的信息，但是因为+1不是idempotent，所以需要在信息传输层保证不会失效，另外一种思路就是不管client还是server都维护count的准确信息，用vector lock来保证同步。关键问题是，我不知道这道题是不是key point就是同步问题。

首先，你先不要曲解题目，你**直接把题目翻译为：《设计distributed counting system》就已经走偏了**。从这道题的题面来看，面试官只是要对每个photo有一个counter。这个counter干嘛的呢？你可以理解为 某个photo 被like的数目。**这个和我们在《系统设计班》第一节twitter课上说的，某个post被like，是一样的。**

第一层：

你首先要知道是用denormailze的方法，和photo 一起存在一起，这样不用去数据库里数like。所以可能考察的就是，数据库的存放方法，服务器端用memcached或者任何cache去存储，访问都是找cache，实在是太大的数据量，才会考虑分布式。+1 分

第二层：

你知道这玩意儿不能每次去数据库查，得cache。 +0.5分

第三层：

这玩意儿一直在更新，被写很多次，你知道必须一直保持这个数据在cache里，不能invalidate。 +0.5 分

第四层：

你知道怎么让数据库和cache保持一致性 +2分

第五层：

你知道 cache 里如果没有了，怎么避免数据库被冲垮（memcache lease get) +2 分

第六层：

一个小的优化，如果这个数据很hot，可以在server内部开一个小cache，只存及其hot的数据。+2分

How does the lease token solve the stale sets problem in Facebook's memcached servers?
https://www.quora.com/How-does-the-lease-token-solve-the-stale-sets-problem-in-Facebooks-memcached-servers
The reason you're not understanding how this works is because you're making a small but important incorrect assumption. You write:

> Suppose Client A tried to get key K, but it's a miss. It also gets lease token L1. At the same time, Client B also tries to fetch key K, it's again, a miss. B gets lease token L2 > L1. They both need to fetch K from DB.

However, this will not happen because if the memcached server has recently given a lease token out, it will not give out another. In other words, **B does not get a lease token in this scenario**. Instead what B receives is a *hot miss* result which tells client B that another client (namely client A) is already headed to the DB for the value, and that client B will have to try again later.

So, leases work in practice because only one client is headed to the DB at any given time for a particular value in the cache.

Furthermore, when a new delete comes in, memcached will know that the next *lease-set* (the one from client A) is already stale, so it will accept the value (because it's newer) but it will mark it as stale and the next client to ask for that value will get a new lease token, and clients after that will once again get hot miss results.

Note also that *hot miss* results include the latest stale value, so a client can make the decision to go forth with slightly stale data, or try again (via exponential backoff) to get the most up-to-date data possible.

https://www.jiuzhang.com/qa/645/
针对同一条记录发生大量并发写的优化问题
一般的方法有：Write back cache。大概的意思就是 Client 只负责写给cache，cache自己去负责写给数据库，但不是每条都写回数据库，隔一段时间写回去。

其实对同一条记录发生大量并发写的情况是很少的，即便Facebook这样的级别，最挑战的并不是短时间很多写，而是短时间很多读，因为只要是给人用的东西，一定是读多于写。短时间很多人读，而此时正好cache里又被没有这个数据，这才是最头疼的。Facebook为了解决这个问题，拓展了memcached，增加了一个叫做 lease-get 的方法。具体原理和实现细节，Google搜索 “Facebook Memcached” :

https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/
http://www.cs.bu.edu/~jappavoo/jappavoo.github.com/451/papers/memcache-fb.pdf

facebook 有两个机制：
1.lease机制， 保证每个时刻每个键只能是拥有唯一一个有效leaseId的客户端写入,《Scaling Memcache at Facebook》。quora有个问答，评论里是fb的相关工程师，简洁明了，包括解释了delete的具体操作：https://www.quora.com/How-does-the-lease-token-solve-the-stale-sets-problem-in-Facebooks-memcached-servers/answer/Ryan-McElroy
2.write-through cache,《TAO: Facebook's Distributed Data Store for the Social Graph》。

我在memcached里已经看不到lease了，但能看到CAS(compare&swap)，请问这是lease的改良品种吗

https://www.jiuzhang.com/qa/1400/
大概就是，memcache发现短时间很多请求通过 lease_get 操作访问同一个key的时候，会让后面来的lease_get 先等着。等到cache里有数据了才返回，而不是直接返回cache里没有。memcache会让第一个least_get 返回失败查询，然后这个lease_get 会去负责回填数据给 cache。

http://abineshtd.blogspot.com/2013/04/notes-on-scaling-memcache-at-facebook.html
https://stackoverflow.com/questions/42417342/does-redis-support-udp-between-server-and-client
No, the Redis protocol, RESP, is TCP based:

> ## Networking layer
>
> A client connects to a Redis server creating a TCP connection to the port 6379.
>
> While RESP is technically non-TCP specific, in the context of Redis the protocol is only used with TCP connections (or equivalent stream oriented connections like Unix sockets).

https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/
One technique that is widely employed is to “warm the cache”. At its most basic level, we run a query that touches each node and relationship in the graph.
https://unix.stackexchange.com/questions/122362/what-does-it-mean-by-cold-cache-and-warm-cache-concept

https://newspaint.wordpress.com/2013/07/12/avoiding-thundering-herd-in-memcached/
A common problem with sites that use memcached is the worry that when memcached is restarted and the cache is empty then a highly-scaled application will hit the cold cache and find it empty and then many threads will proceed to all simultaneously make heavy computed lookups (typically in a database) bringing the whole site to a halt. This is called the “Thundering Herd” problem.

There are two worries about “Thundering Herd”. The first, and considered by this article, is where you may have a complex and expensive database query that is used to calculate a cached value; you don’t want 100 users simultaneously trying to access this value soon after you start your application resulting in that complex database query being executed 100 times when a single execution will do the job. Better to wait for that single query to complete, populate the cache, then all the other users can get the response near instantly afterwards. The second worry is that an empty cache will need filling from many different types of queries – this article does not help with that problem – but then pre-filling a cache could be difficult (if which needed cached values are not determinable ahead of time) and will take time in such a situation nonetheless.

A common recommendation to avoid this is to create “warm-up” scripts that pre-fill memcached before letting the application loose at it.

I think there’s another way. Using access controls via memcached itself. Because the locks should be cheap and access fast (compared to doing a database lookup). You can build these techniques into your application and forget about Thundering Herd.

This works by locking memcached keys using another memcached key. And putting your application to sleep (polling) until the lock is released if another process has it. This way if two clients attempt to access the same key – then one can discover it is missing/empty, do the expensive computation once, and then release the lock – the other client will wake up and use the pre-computed key.

A remedial action could be to specify a psuedo-random number as the sleep time – thus greatly increasing the probability that different threads will wake up at different times reducing the possibility of contention with another.. if this worries you enough.

https://www.educative.io/collection/page/5668639101419520/5649050225344512/5643440998055936
EhCache: refresh ahead
https://www.ehcache.org/apidocs/2.8.4/net/sf/ehcache/constructs/refreshahead/RefreshAheadCache.html
A cache decorator which implements read ahead refreshing. Read ahead occurs when a cache entry is accessed prior to its expiration, and triggers a reload of the value in the background.
A significant attempt is made to ensure only one node of the cache works on a specific key at a time. There is no guarantee that every triggered refresh ahead case will be processed. As the maximum number of backlog entries is reached, refresh ahead requests will be dropped silently.
https://github.com/svn2github/ehcache/blob/master/trunk/ehcache/ehcache-core/src/test/resources/ehcache-refresh-ahead-simple.xml

https://stackoverflow.com/questions/32214191/eagerly-repopulate-ehcache-instead-of-waiting-for-a-read
Eagerly repopulate EhCache instead of waiting for a read
You will need two things in order to make this work with Ehcache:

- Use a cache loader - that is move to a cache read-through pattern. This is required as otherwise Ehcache has no idea how to get to the data mapped to a key.
- Configure scheduled refresh - this works by launching a quartz scheduler instance.

http://ww1.terracotta.org/documentation/4.1/bigmemorymax/api/refresh-ahead

https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#COHDG5177

Posted by Jeffery at 9:07 AM

Labels: Cache, System Design

No comments:

Post a Comment

Enter your comment...

Comment as: anilchowdhury1 ▾ Sign out

Publish Preview Notify me

Links to this post

Create a Link

Subscribe to: Post Comments (Atom)

Labels

Review (572) System Design (334) System Design - Review (198) Java (189) Coding (75) Interview-System Design (65) Interview (63) Book Notes (59) Coding - Review (59) to-do (45) Linux (41) Knowledge (39) Interview-Java (35) Knowledge - Review (32) Database (31) Design Patterns (31) Big Data (29) Product Architecture (28) MultiThread (27) Soft Skills (27) Concurrency (26) Cracking Code Interview (26) Miscs (25) Distributed (24) OOD Design (24) Career (22) Google (21) Interview - Review (21) Java - Code (21) Operating System (21) Interview Q&A (20) System Design - Practice (20) Tips (19) Algorithm (17) Company - Facebook (17) Security (17) How to Ace Interview (16) Brain Teaser (14) Linux - Shell (14) Redis (14) Tools (14) Code Quality (13) Search (13) Spark (13) Spring (13) Company - LinkedIn (12) How to (12) Interview-Database (12) Interview-Operating System (12) Solr (12) Architecture Principles (11) Testing (11) Resource (10) Amazon (9) Cache (9) Git (9) Interview - MultiThread (9) Scalability (9) Trouble Shooting (9) Web Dev (9) Architecture Model (8) Better Programmer (8) Cassandra (8) Company - Uber (8) Java67 (8) Math (8) OO Design principles (8) SOLID (8) Design (7) Interview Corner (7) JVM (7) Java Basics (7) Kafka (7) Mac (7) Machine Learning (7) NoSQL (7) C++ (6) File System (6) Highscalability (6) How to Better (6) Network (6) Restful (6) CareerCup (5) Code Review (5) Hash (5) How to Interview (5) JDK Source Code (5) JavaScript (5) Leetcode (5) Must Known (5) Python (5)

Popular Posts

Archives

Design a chess game using OO principles | Runhe Tian Coding Practice
http://k2code.blogspot.com/2014/03/design-chess-game-using-oo-principles.html http://swcodes.blogspot.in/2012/09/chess-game-design.html ...

thought-works: Object Oriented design for Elevator in a multi-storied apartment
thought-works: Object Oriented design for Elevator in a multi-storied apartment A typical lift has buttons(Elevator buttons) inside the ca...

How to design a tiny URL or URL shortener?
Related: http://massivetechinterview.blogspot.com/2015/06/n00tc0d3r.html https://puncsky.com/hacking-the-software-engineer-interview#desi...

Google Map Architecture
http://all-things-spatial.blogspot.com/2009/06/ingenuity-of-google-map-architecture.html The traditional way to publish maps over the Inte...

[CC150v5] 8.4 Design a Parking Lot - Shuatiblog.com
lintcode 498. Parking Lot 在职刷题 + System Design + 面试准备的路上 How to design parking system: 需要以parking lot为视角进行操作, 然后主体主要有两个，vehicle和parking s...

Google Calendar Architecture
如何设计类似Google Calendar的系统 http://computer.howstuffworks.com/internet/basics/google-calendar.htm You can choose to view the calendar by day...

Design a chat server | Hello World
https://www.interviewbit.com/problems/design-messenger/ Q: What is the scale that we are looking at? A: Let's assume the scale of ...

Design a news feed system
http://blog.gainlo.co/index.php/2016/03/29/design-news-feed-system-part-1-system-design-interview-questions/ In fact, there are a bunch of...

Design Hit Counter - how to count number of requests in last second, minute and hour - Stack Overflow
http://massivealgorithms.blogspot.com/2016/08/leetcode-362-design-hit-counter.html 统计网站最近5分钟的访问次数
https://www.1point3acres.com/bbs/forum....