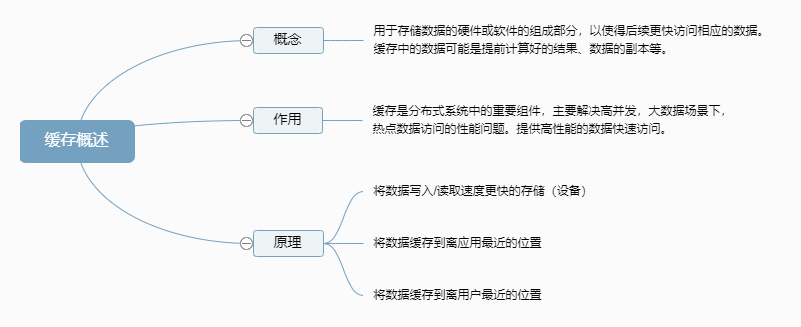
# 概述



# 分类

缓存主要分为四类：

1. CDN缓存
2. 反向代理缓存
3. 本地应用缓存
4. 分布式缓存

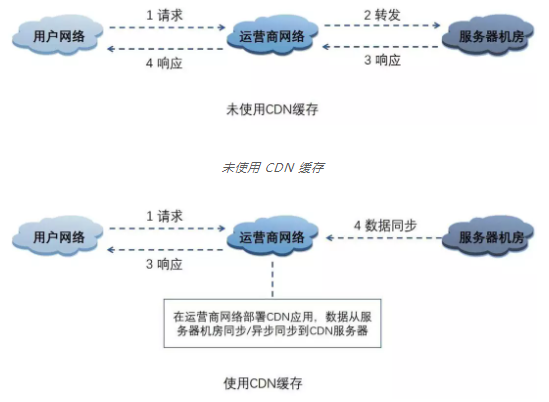
## CDN缓存

CDN(Content Delivery Network 内容分发网络)的基本原理是广泛采用各种缓存服务器，将这些缓存服务器分布到用户访问相对集中的地区或网络中。

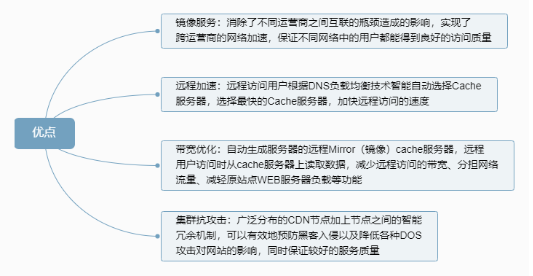
在用户访问网站时，利用全局负载技术将用户的访问指向距离最近的工作正常的缓存服务器上，由缓存服务器直接响应用户请求。

应用场景：主要**缓存静态资源**，例如图片，视频。

CDN 缓存应用如下图：



CDN缓存优点如下图：



## 反向代理缓存

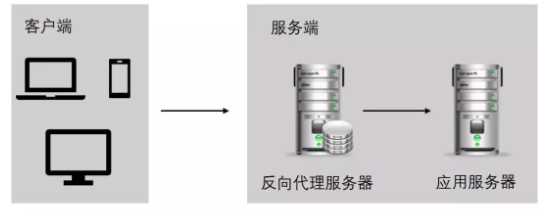
反向代理位于应用服务器机房，处理所有对Web服务器的请求。

如果用户请求的页面在代理服务器上有缓冲的话，代理服务器直接将缓冲内容发送给用户。

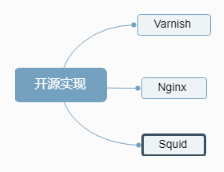
如果没有缓冲则先向Web服务器发出请求，取回数据，本地缓存后再发送给用户。通过降低向Web服务器的请求数，从而降低了Web服务器的负载。

应用场景：一般只缓存体积较小静态文件资源，如css、js、图片。

反向代理缓存应用如下图：



开源实现如下图：



## 本地应用缓存

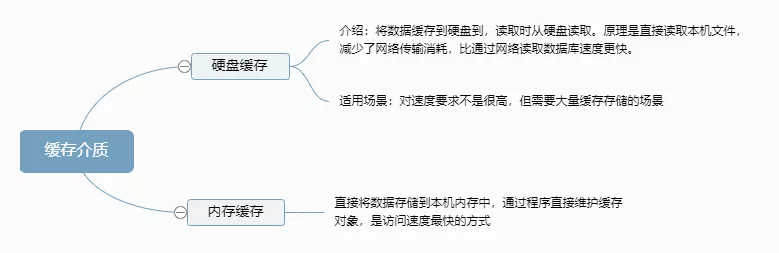
指的是在应用中的缓存组件，其最大的优点是应用和Cache是在同一个进程内部，请求缓存非常快速，没有过多的网络开销等。

在单应用不需要集群支持或者集群情况下各节点无需互相通知的场景下使用本地缓存较合适。

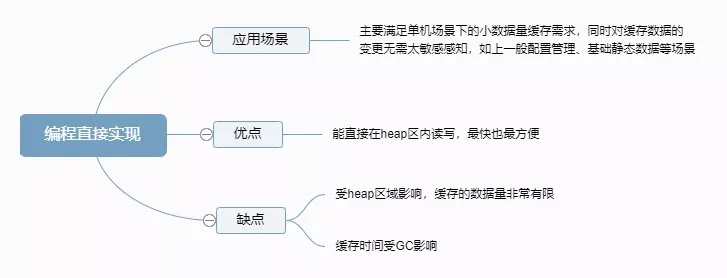
同时，它的缺点也是应为缓存跟应用程序耦合，多个应用程序无法直接的共享缓存，各应用或集群的各节点都需要维护自己的单独缓存，对内存是一种浪费。

**应用场景：**缓存字典等常用数据。

缓存介质如下图所示：



编程直接实现如下图：



### Ehcache

基本介绍：Ehcache是一种基于标准的开源缓存，可提高性能，卸载数据库并简化可伸缩性。

它是使用最广泛的基于Java的缓存，因为它功能强大，经过验证，功能齐全，并与其他流行的库和框架集成。

Ehcache可以从进程内缓存扩展到使用TB级缓存的混合进程内/进程外部署。

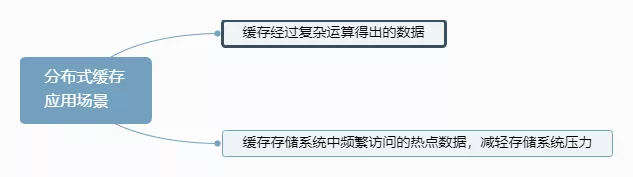
### Guava Cache

基本介绍：Guava Cache是Google开源的Java重用工具集库Guava里的一款缓存工具。

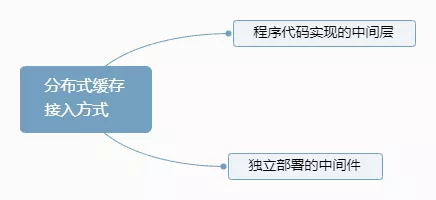
## 分布式缓存

指的是与应用分离的缓存组件或服务，其最大的优点是自身就是一个独立的应用，与本地应用隔离，多个应用可直接的共享缓存。

分布式缓存的主要应用场景如下图：



分布式缓存的主要接入方式如下图：



### Memcached

#### 概述

Memcached是一个高性能，分布式内存对象缓存系统，通过在内存里维护一个统一的巨大的Hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。

简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。

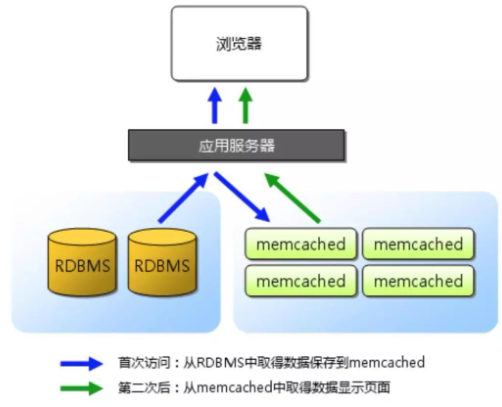
#### 特点

Memcached的特点如下图：



#### 架构

Memcached的基本架构如下图：



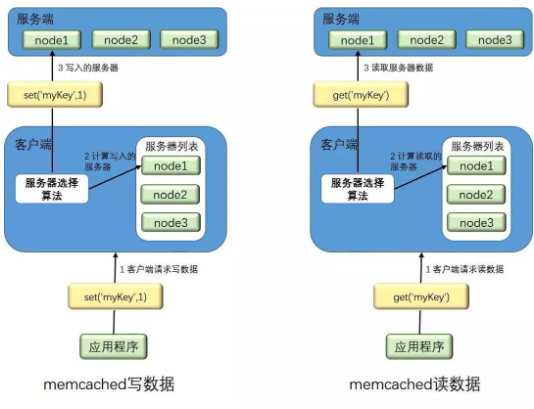
**缓存数据过期策略：**LRU（最近最少使用）到期失效策略，在Memcached内存储数据项时，可以指定它在缓存的失效时间，默认为永久。

当Memcached服务器用完分配的内存时，失效的数据被首先替换，然后是最近未使用的数据。

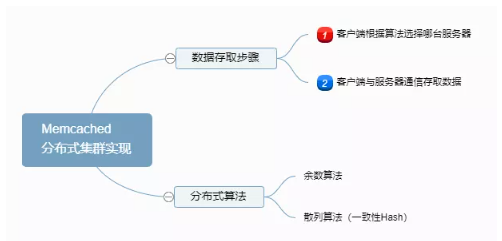
**数据淘汰内部实现：**懒淘汰机制为每次往缓存放入数据的时候，都会存一个时间，在读取的时候要和设置的时间做TTL比较来判断是否过期。

**分布式集群实现：**服务端并没有“分布式”功能。每个服务器都是完全独立和隔离的服务。Memcached的分布式，是由客户端程序实现的。

**数据读写流程图：**



Memcached分布式集群实现：



### Redis



# 缓存问题

缓存带来的复杂度问题，常见的问题主要包括如下几点：

数据一致性

缓存穿透

缓存雪崩

缓存高可用

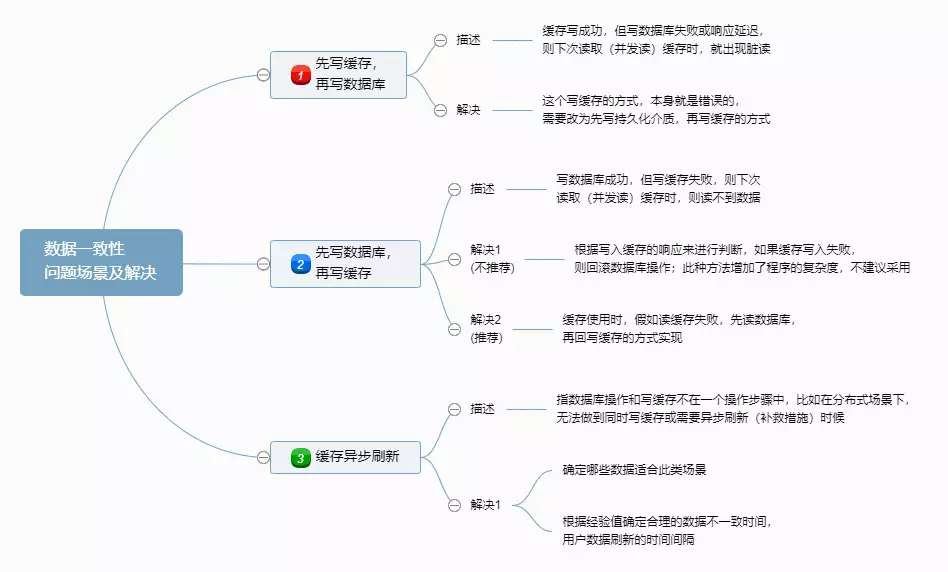
缓存热点

## 缓存一致性

缓存一致性是指业务在引入分布式缓存系统后，业务对数据的更新除了要更新存储以外还需要同时更新缓存，对两个系统进行数据更新就要先解决分布式系统中的隔离性和原子性难题。目前大多数业务在引入分布式缓存后都是通过牺牲小概率的一致性来保障业务性能，因为要在业务层严格保障数据的一致性，代价非常高，业务引入分布式缓存主要是为了解决性能问题，所以在性能和一致性面前，通常选择牺牲小概率的一致性来保障业务性能。

因为缓存属于持久化数据的一个副本，因此不可避免的会出现数据不一致问题，导致脏读或读不到数据的情况。

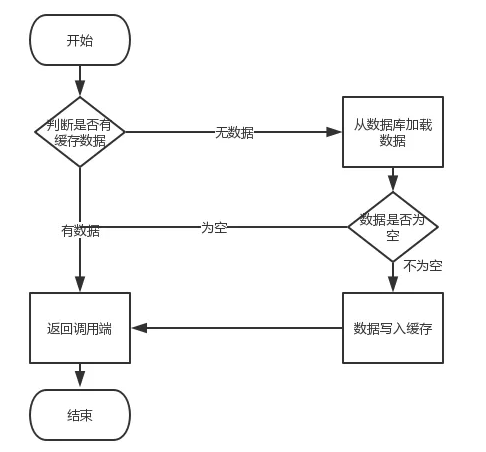
数据不一致，一般是因为网络不稳定或节点故障导致问题出现的常见 3 个场景以及解决方案：



说明：在TDSQL3.0的统计信息加载中，就遇到多个SQLEngine的缓存一致性问题，这个是通过增加广播ANALYZE TABLE RELOAD实现同步的。

### 背景

缓存由于其高并发和高性能的特性，已经在项目中被广泛使用。在读取缓存方面，都是按照下图的流程来进行业务操作：



但是在更新缓存方面，对于更新完数据库，是更新缓存呢，还是删除缓存。又或者是先删除缓存，再更新数据库，其实大家存在很大的争议。

### 缓存更新策略

从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。这种方案下，我们可以对存入缓存的数据设置过期时间，所有的写操作以数据库为准，对缓存操作只是尽最大努力即可。也就是说如果数据库写成功，缓存更新失败，那么只要到达过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。因此，接下来讨论的思路不依赖于给缓存设置过期时间这个方案。

在这里，我们讨论三种更新策略：

1. 先更新数据库，再更新缓存
2. 先删除缓存，再更新数据库
3. 先更新数据库，再删除缓存

#### 先更新数据库，再更新缓存

这套方案，大家是普遍反对的。为什么呢？有如下两点原因。

**原因一（线程安全角度）**

同时有请求A和请求B进行更新操作，那么会出现

（1）线程A更新了数据库

（2）线程B更新了数据库

（3）线程B更新了缓存

（4）线程A更新了缓存

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据，因此不考虑。

**原因二（业务场景角度）**

有如下两点：

（1）如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。

（2）如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

接下来讨论的就是争议最大的，先删缓存，再更新数据库。还是先更新数据库，再删缓存的问题。

#### 先删缓存，再更新数据库

该方案会导致不一致的原因是。同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么会出现如下情形:

（1）请求A进行写操作，删除缓存

（2）请求B查询发现缓存不存在

（3）请求B去数据库查询得到旧值

（4）请求B将旧值写入缓存

（5）请求A将新值写入数据库

上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

那么，如何解决呢？采用**延时双删策略**

伪代码如下

public void write(Stringkey,Objectdata){

redis.delKey(key);

db.updateData(data);

Thread.sleep(1000);

redis.delKey(key);

}

转化为中文描述就是

（1）先淘汰缓存

（2）再写数据库（这两步和原来一样）

（3）休眠1秒，再次淘汰缓存

这么做，可以将1秒内所造成的缓存脏数据，再次删除。

那么，这个1秒怎么确定的，具体该休眠多久呢？

针对上面的情形，读者应该自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

如果你用了mysql的读写分离架构怎么办？

在这种情况下，造成数据不一致的原因如下，还是两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。

（1）请求A进行写操作，删除缓存

（2）请求A将数据写入数据库了，

（3）请求B查询缓存发现，缓存没有值

（4）请求B去从库查询，这时，还没完成主从同步，因此查询到的是旧值

（5）请求B将旧值写入缓存

（6）数据库完成主从同步，从库变为新值

上述情形，就是数据不一致的原因。还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。

采用这种同步淘汰策略，吞吐量降低怎么办？

那就将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，再返回。这么做，加大吞吐量。

第二次删除,如果删除失败怎么办？

这是个非常好的问题，因为第二次删除失败，就会出现如下情形。还是有两个请求，一个请求A进行更新操作，另一个请求B进行查询操作，为了方便，假设是单库：

（1）请求A进行写操作，删除缓存

（2）请求B查询发现缓存不存在

（3）请求B去数据库查询得到旧值

（4）请求B将旧值写入缓存

（5）请求A将新值写入数据库

（6）请求A试图去删除请求B写入对缓存值，结果失败了。

这也就是说。如果第二次删除缓存失败，会再次出现缓存和数据库不一致的问题。

如何解决呢？

具体解决方案，且看博主对第(3)种更新策略的解析。

#### 先更新数据库，再删缓存

首先，先说一下。老外提出了一个缓存更新套路，名为《Cache-Aside pattern》。其中就指出

失效：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从cache中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。

另外，知名社交网站facebook也在论文《Scaling Memcache at Facebook》中提出，他们用的也是先更新数据库，再删缓存的策略。

这种情况不存在并发问题么？

不是的。假设这会有两个请求，一个请求A做查询操作，一个请求B做更新操作，那么会有如下情形产生

（1）缓存刚好失效

（2）请求A查询数据库，得一个旧值

（3）请求B将新值写入数据库

（4）请求B删除缓存

（5）请求A将查到的旧值写入缓存

如果发生上述情况，确实是会发生脏数据。

然而，发生这种情况的概率又有多少呢？

发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。

假设，有人非要抬杠，有强迫症，一定要解决怎么办？

如何解决上述并发问题？

首先，给缓存设有效时间是一种方案。其次，采用策略（2）里给出的异步延时删除策略，保证读请求完成以后，再进行删除操作。

还有其他造成不一致的原因么？

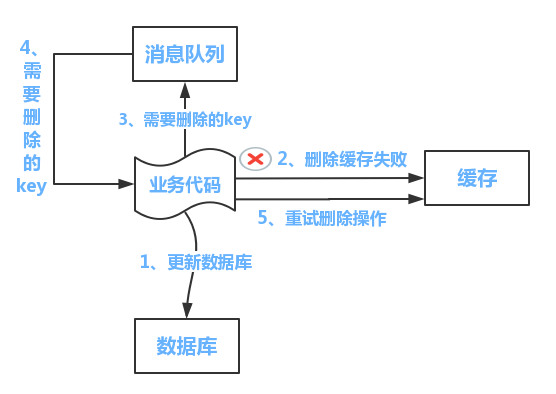
有的，这也是缓存更新策略（2）和缓存更新策略（3）都存在的一个问题，如果删缓存失败了怎么办，那不是会有不一致的情况出现么。比如一个写数据请求，然后写入数据库了，删缓存失败了，这会就出现不一致的情况了。这也是缓存更新策略（2）里留下的最后一个疑问。

### 解决方案

如何解决？

提供一个保障的重试机制即可，这里给出两套方案。

方案一：



流程如下所示

（1）更新数据库数据；

（2）缓存因为种种问题删除失败

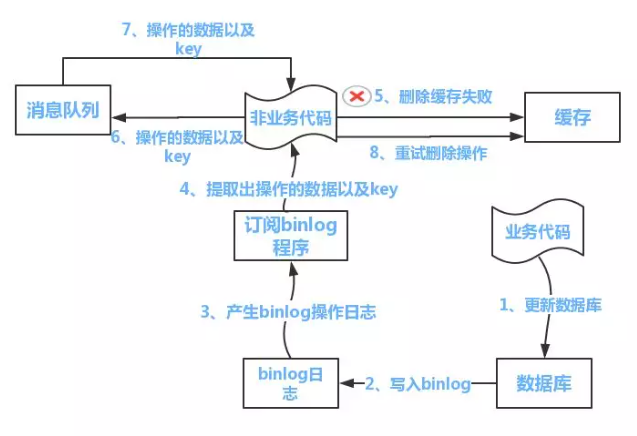
（3）将需要删除的key发送至消息队列

（4）自己消费消息，获得需要删除的key

（5）继续重试删除操作，直到成功

然而，该方案有一个缺点，对业务线代码造成大量的侵入。于是有了方案二，在方案二中，启动一个订阅程序去订阅数据库的binlog，获得需要操作的数据。在应用程序中，另起一段程序，获得这个订阅程序传来的信息，进行删除缓存操作。

方案二：



（1）更新数据库数据

（2）数据库会将操作信息写入binlog日志当中

（3）订阅程序提取出所需要的数据以及key

（4）另起一段非业务代码，获得该信息

（5）尝试删除缓存操作，发现删除失败

（6）将这些信息发送至消息队列

（7）重新从消息队列中获得该数据，重试操作。

说明：上述的订阅binlog程序在mysql中有现成的中间件叫canal，可以完成订阅binlog日志的功能。至于oracle中，目前不知道有没有现成中间件可以使用。如果对一致性要求不是很高，直接在程序中另起一个线程，每隔一段时间去重试即可。

## 缓存穿透

正常情况下，我们去查询数据都是存在。那么请求去查询一条压根数据库中根本就不存在的数据，也就是缓存和数据库都查询不到这条数据，但是请求每次都会打到数据库上面去。

这种**查询不存在数据**的现象我们称为缓存穿透。

### 问题

试想一下，如果有黑客会对你的系统进行攻击，拿一个不存在的id去查询数据，会产生大量的请求到数据库去查询。可能会导致你的数据库由于压力过大而宕掉。

### 解决

常见缓存穿透、击穿、雪崩的解决方案：

1. 直接缓存NULL值
2. 限流
3. 缓存预热
4. 分级缓存
5. 缓存永远不过期

缓存污染的场景我们目前还没有发现较好的解决方案，但是在空数据查询问题上我们可以改造业务，通过以下方式防止缓存击穿：

1、通过bloomfilter记录key是否存在，从而避免无效Key的查询；

2、在Redis缓存不存在的Key，从而避免无效Key的查询；

#### 缓存空值

之所以会发生穿透，就是因为缓存中没有存储这些空数据的key。从而导致每次查询都到数据库去了。

那么我们就可以为这些key设置的值设置为null丢到缓存里面去。后面再出现查询这个key的请求的时候，直接返回null ,就不用在到数据库中去走一圈了。但是别忘了设置过期时间。

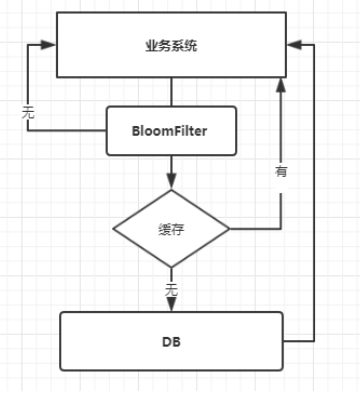
应对缓存穿透最有效的方法是直接缓存NULL值，但是缓存NULL的时间不能太长，否则NULL数据长时间得不到更新，也不能太短，否则达不到防止缓存击穿的效果。

#### BloomFilter

BloomFilter类似于一个hase set用来判断某个元素（key）是否存在于某个集合中。

这种方式在大数据场景应用比较多，比如Hbase中使用它去判断数据是否在磁盘上。还有在爬虫场景判断url是否已经被爬取过。

这种方案可以加在第一种方案中，在缓存之前在加一层BloomFilter，在查的时候先去BloomFilter去查询key是否存在，如果不存在就直接返回，存在再走查缓存->查DB。



**优点：**

1、思路简单

2、保证一致性

3、性能强

**缺点：**

1、代码复杂度增大

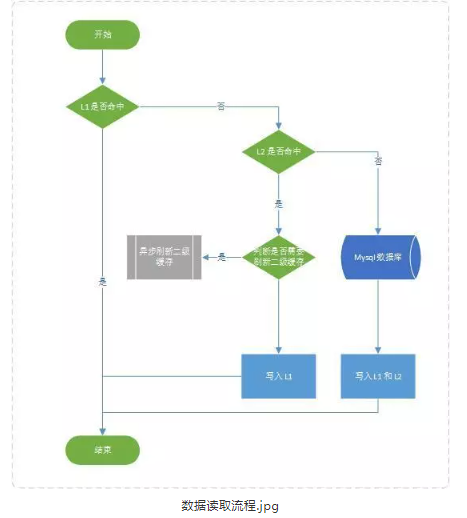
2、需要另外维护一个集合来存放缓存的Key

3、布隆过滤器不支持删值操作

#### 限流

应对缓存穿透的常用方法之一是限流，常见的限流算法有滑动窗口，令牌桶算法和漏桶算法，或者直接使用队列、加锁等，在layering-cache里面主要使用分布式锁来做限流。

layering-cache数据读取流程：



当需要加载缓存的时候，需要获取到锁才有权限到后台去加载缓存数据，否则就会等待（同一个线程循环20次查询缓存，每次等待20毫秒，如果还是没有数据直接去执行被缓存的方法，这个主要是为了防止获取到锁并且去加载缓存的线程出问题，没有返回而导致死锁）。当获取到锁的线程执行完成会将获取到的数据放到缓存中，并且唤醒所有等待线程。

这里需要注意一下让线程等待一定不能用Thread.sleep()，我在使用Spring Redis Cache的时候，发现当并发达到300左右，缓存一旦过期就会引起死锁，原因是使用的是sleep方法来让没有获取到锁的线程等待，当等待的线程很多的时候会产生大量上下文切换，导致获取到锁的线程一直获取不到cpu的执行权，导致死锁。在layering-cache里面，我们使用的是LockSupport.parkNanos方法，它会释放cpu资源, 因为我们使用的是redis分布式锁，所以也不能使用wait-notify机制。

#### 缓存预热

有效应对缓存的击穿和雪崩的方式之一是缓存预加载。

在layering-cache里面二级缓存会配置两个时间，expireTime是缓存的过期时间，preloadTime是缓存的刷新时间（预加载时间）。每次二级缓存被命中都会去检查缓存的过去时间是否小于刷新时间，如果小于就会开启一个异步线程预先去更新缓存，并将新的值放到缓存中，有效的保证了热点数据\*\*"永不过期"\*\*。这里预先更新缓存也是需要加锁的，并不是所有的线程都会落到库上刷新缓存，如果没有获取到锁就直接结束当前线程。

在缓存总量和并发量都很大的时候，这个时候缓存如果同时失效，缓存预热将是一个非常慢长的过程，就比如说服务重启或新上线一个新的缓存。这个时候我们可以采用切流的方式，让缓存慢慢预热，如开始切10%流量，观察没有异常后，再切30%流量，观察没有异常后，再切60%流量，然后全量。这种方式虽然有点繁琐，但是一旦遇到异常我们可以快速的切回流量，让风险可控。

**总结**

总体来说layering-cache在缓存穿透、击穿和雪崩上是以预防为主，补救为辅。而在应对缓存的这些问题上其实也没有一个完全完美的方案，只有最适合自己业务系统的方案。目前如果直接使用layering-cache缓存框架已经基本能应对大部分的缓存问题了。

### 选择

针对于一些恶意攻击，攻击带过来的大量key是不存在的，那么我们采用第一种方案就会缓存大量不存在key的数据。此时我们采用第一种方案就不合适了，我们完全可以先对使用第二种方案进行过滤掉这些key。

**针对这种key异常多，请求重复率比较低的数据，我们就没有必要进行缓存，使用第二种方案直接过滤掉**。

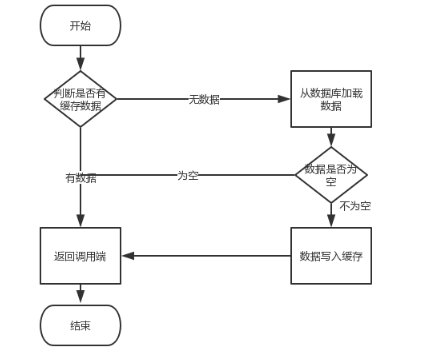
**对于空数据的key有限的，重复率比较高的，我们则可以采用第一种方式进行缓存。**

## 缓存击穿

在平常高并发的系统中，大量的请求同时查询一个key时，此时这个**key正好失效**了，就会导致大量的请求都打到数据库上面去。这种现象我们称为击穿。

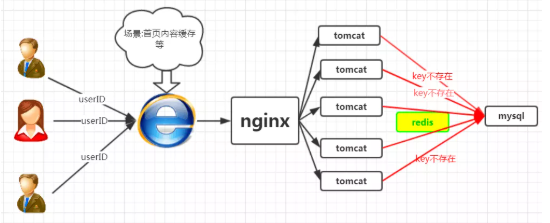
### 背景

从缓存中加载数据的逻辑，如下图所示：



因此，如果黑客每次故意查询一个在缓存内必然不存在的数据，导致每次请求都要去存储层去查询，这样缓存就失去了意义。如果在大流量下数据库可能挂掉。这就是缓存击穿。

场景如下图所示:



我们正常人在登录首页的时候，都是根据userID来命中数据，然而黑客的目的是破坏你的系统，黑客可以随机生成一堆userID,然后将这些请求怼到你的服务器上，这些请求在缓存中不存在，就会穿过缓存，直接怼到数据库上,从而造成数据库连接异常。

缓存击穿是指查询请求没有在缓存层命中而将查询透传到存储DB的问题，当大量的请求发生缓存击穿时，将给存储DB带来极大的访问压力，甚至导致DB过载拒绝服务。空数据查询(黑客攻击)和缓存污染（网络爬虫）是常见的引发缓存击穿的原因。什么是空数据查询？空数据查询通常指攻击者伪造大量不存在的数据进行访问（比如不存在的商品信息、用户信息）。缓存污染通常指在遍历数据等情况下冷数据把热数据驱逐出内存，导致缓存了大量冷数据而热数据被驱逐。缓存污染的场景我们目前还没有发现较好的解决方案，但是在空数据查询问题上我们可以改造业务，通过以下方式防止缓存击穿：

1. 通过bloomfilter记录key是否存在，从而避免无效Key的查询；
2. 在Redis缓存不存在的Key，从而避免无效Key的查询；

### 问题

会造成某一时刻数据库请求量过大，压力剧增，甚至导致DB过载拒绝服务。

空数据查询(黑客攻击)和缓存污染（网络爬虫）是常见的引发缓存击穿的原因。什么是空数据查询？空数据查询通常指攻击者伪造大量不存在的数据进行访问（比如不存在的商品信息、用户信息）。缓存污染通常指在遍历数据等情况下冷数据把热数据驱逐出内存，导致缓存了大量冷数据而热数据被驱逐。

### 解决

在这里给出三套解决方案，根据项目中的实际情况，选择使用。

讲下述三种方案前，先概述redis的setnx方法

**SETNX key value**

将key的值设为value ，当且仅当key不存在。

若给定的key已经存在，则 SETNX不做任何动作。

SETNX是『SET if Not eXists』(如果不存在，则SET)的简写。

**可用版本：**>= 1.0.0

**时间复杂度：**O(1)

**返回值：**设置成功，返回1。设置失败，返回0。

效果如下：

redis> EXISTS job # job 不存在

(integer) 0

redis> SETNX job "programmer" # job 设置成功

(integer) 1

redis> SETNX job "code-farmer" # 尝试覆盖 job ，失败

(integer) 0

redis> GET job # 没有被覆盖

"programmer"

#### 互斥锁

上面的现象是多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它。**其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后做缓存**。后面的线程进来发现已经有缓存了，就直接走缓存。

该方法是比较普遍的做法，即，在根据key获得的value值为空时，先锁上，再从数据库加载，加载完毕，释放锁。若其他线程发现获取锁失败，则睡眠50ms后重试。

**至于锁的类型，单机环境用并发包的Lock类型就行，集群环境则使用分布式锁( redis的setnx)。**

集群环境的redis的代码如下所示:

String get(String key) {

String value = redis.get(key);

if (value == null) {

if (redis.setnx(key\_mutex, "1")) {

// 3 min timeout to avoid mutex holder crash

redis.expire(key\_mutex, 3 \* 60)

value = db.get(key);

redis.set(key, value);

redis.delete(key\_mutex);

} else {

//其他线程休息50毫秒后重试

Thread.sleep(50);

get(key);

}

}

}

**优点：**

1、思路简单

2、保证一致性

**缺点：**

1、代码复杂度增大

2、存在死锁的风险

#### 异步构建缓存

在这种方案下，构建缓存采取异步策略，会从线程池中取线程来异步构建缓存，从而不会让所有的请求直接怼到数据库上。该方案redis自己维护一个timeout，当timeout小于System.currentTimeMillis()时，则进行缓存更新，否则直接返回value值。

集群环境的redis代码如下所示:

String get(final String key) {

V v = redis.get(key);

String value = v.getValue();

long timeout = v.getTimeout();

if (v.timeout <= System.currentTimeMillis()) {

// 异步更新后台异常执行

threadPool.execute(new Runnable() {

public void run() {

String keyMutex = "mutex:" + key;

if (redis.setnx(keyMutex, "1")) {

// 3 min timeout to avoid mutex holder crash

redis.expire(keyMutex, 3 \* 60);

String dbValue = db.get(key);

redis.set(key, dbValue);

redis.delete(keyMutex);

}

}

});

}

return value;

}

**优点：**

性价最佳，用户无需等待

**缺点：**

无法保证缓存一致性

#### 布隆过滤器

redis伪代码如下所示

String get(String key) {

String value = redis.get(key);

if (value == null) {

if(!bloomfilter.mightContain(key)){

return null;

}else{

value = db.get(key);

redis.set(key, value);

}

}

return value；

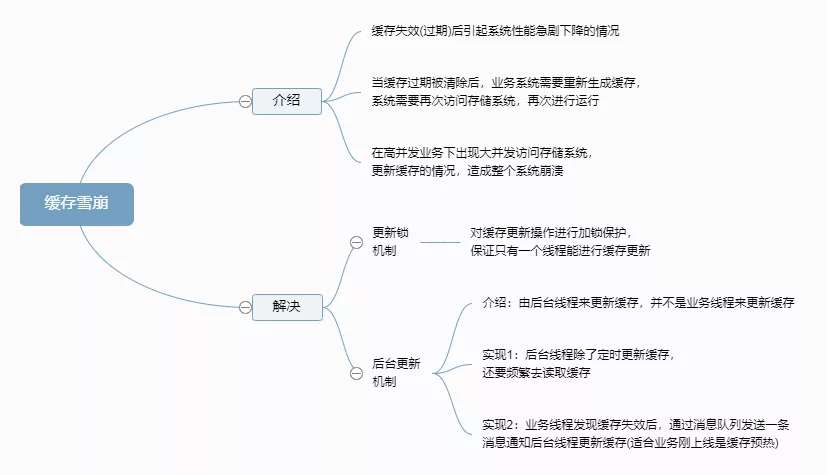
**优点：**

1. 思路简单
2. 保证一致性
3. 性能强

**缺点：**

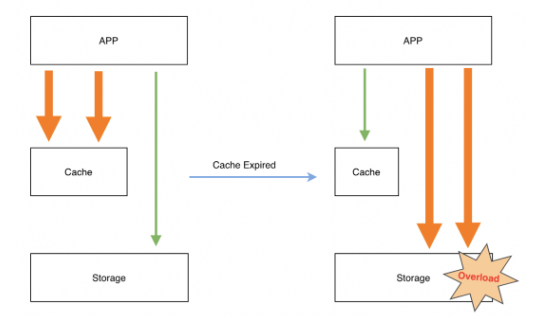
1. 代码复杂度增大
2. 需要另外维护一个集合来存放缓存的Key
3. 布隆过滤器不支持删值操作

## 缓存雪崩



缓存雪崩是指由于大量的热数据设置了相同或接近的过期时间，导致缓存在某一时刻密集失效，大量请求全部转发到DB，或者是某个冷数据瞬间涌入大量访问，这些查询在缓存MISS后，并发的将请求透传到DB，DB瞬时压力过载从而拒绝服务。

目前常见的预防缓存雪崩的解决方案，主要是通过对key的TTL时间加随机数，打散key的淘汰时间来尽量规避，但是不能彻底规避。



缓存的情况是说，当某一时刻发生**大规模**的缓存**失效**的情况。比如你的缓存服务宕机了，会有大量的请求进来直接发送到DB，结果就是DB宕机。

注：缓存击穿是数据不存在，缓存雪崩是数据失效，二者本质不同。

### 问题

缓存雪崩是指由于大量的热数据设置了**相同或接近的过期时间**，导致缓存在某一时刻密集失效，大量请求全部转发到 DB，或者是某个冷数据瞬间涌入大量访问，这些查询在缓存 MISS 后，并发的将请求透传到 DB，DB 瞬时压力过载从而拒绝服务。

### 解决

**常见解决方案：**

直接缓存NULL值

限流

缓存预热

分级缓存

缓存永远不过期

**事前：使用集群缓存，保证缓存服务的高可用**

这种方案就是在发生雪崩前对缓存集群实现高可用，如果是使用Redis，可以使用主从+哨兵，Redis Cluster来避免Redis全盘崩溃的情况。

**事中：使用ehcache本地缓存+Hystrix限流&降级 ,避免MySQL被打死的情况发生。**

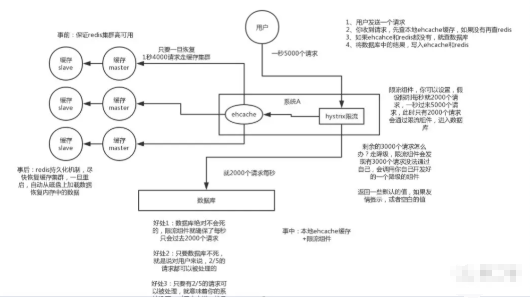
使用ehcache本地缓存的目的也是考虑在Redis Cluster完全不可用的时候ehcache本地缓存还能够支撑一阵。

使用Hystrix进行限流 & 降级 ，比如一秒来了5000个请求，我们可以设置假设只能有一秒2000个请求能通过这个组件，那么其他剩余的 3000 请求就会走限流逻辑，然后去调用我们自己开发的降级组件（降级）。比如设置的一些默认值呀之类的。以此来保护最后的MySQL不会被大量的请求给打死。

**事后：开启Redis持久化机制，尽快恢复缓存集群**

一旦重启，就能从磁盘上自动加载数据恢复内存中的数据。

防止雪崩方案如下图所示：



#### 异步构建缓存

在这种方案下，构建缓存采取异步策略，会从线程池中取线程来异步构建缓存，从而不会让所有的请求直接怼到数据库上。该方案redis自己维护一个timeout，当timeout小于System.currentTimeMillis()时，则进行缓存更新，否则直接返回value值。

**优点：**

性价最佳，用户无需等待

**缺点：**

无法保证缓存一致性

## 缓存高可用

缓存是否高可用，需要根据实际的场景而定，并不是所有业务都要求缓存高可用，需要结合具体业务，具体情况进行方案设计，例如临界点是否对后端的数据库造成影响。

主要解决方案：

1. 分布式：实现数据的海量缓存
2. 复制：实现缓存数据节点的高可用。

## 热点数据集中失效

一些特别热点的数据，高并发访问同一份缓存数据，导致缓存服务器压力过大。

### 问题

我们在设置缓存的时候，一般会给缓存设置一个失效时间，过了这个时间，缓存就失效了。对于一些热点的数据来说，当缓存失效以后会存在大量的请求过来，然后打到数据库去，从而可能导致数据库崩溃的情况。

### 解决

解决：复制多份缓存副本，把请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压力。

#### 设置失效时间

1、设置不同的失效时间

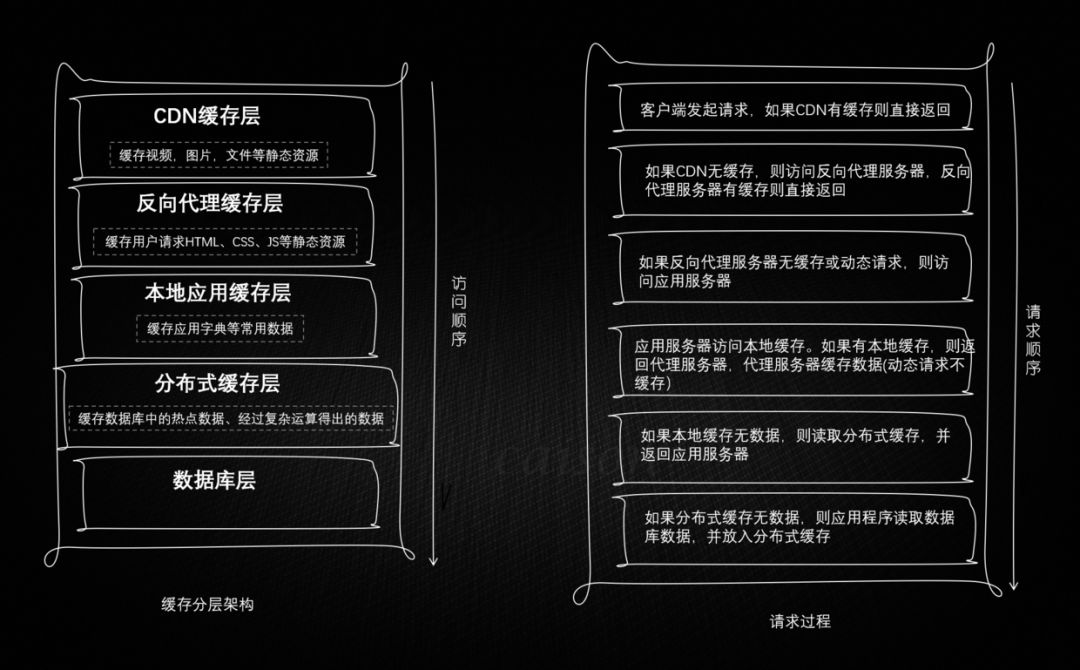
为了避免这些热点的数据集中失效，那么我们在设置缓存过期时间的时候，我们让他们失效的时间错开。比如在一个基础的时间上加上或者减去一个范围内的随机值。

#### 互斥锁

2、互斥锁

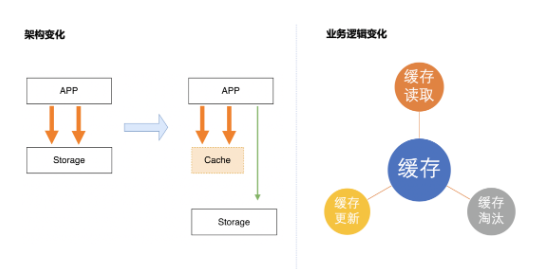
结合上面的击穿的情况，在第一个请求去查询数据库的时候对他加一个互斥锁，其余的查询请求都会被阻塞住，直到锁被释放，从而保护数据库。但是也是由于它会阻塞其他的线程，此时系统吞吐量会下降。需要结合实际的业务去考虑是否要这么做。

# 分层缓存架构设计



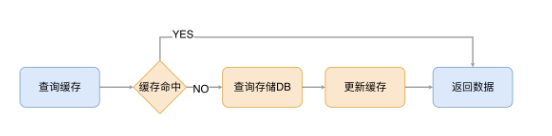
# 传统分布式缓存方案

在引入分布式缓存后，我们的业务架构由原有两层架构（应用+数据库）变成了三层架构（应用+缓存+存储），缓存层缓存热数据，存储层负责全量数据持久化存储。存储架构的变化要求业务对数据的存取逻辑进行相应调整，而且这个调整是巨大的。在缓存系统的选择上，常见的缓存数据库包括Memcached、Redis，目前使用最广泛的是Redis，存储数据常见的包括关系型数据库MySQL、PG、Oreacle、SQLServer等，NoSQL数据库MongoDB、Hbase等。在引入分布式缓存后，业务逻辑需要做三个点的变化，缓存读取、缓存更新、缓存淘汰。



## 缓存读取

引入缓存层后，读数据就变得不是那么简单直接了，APP 需要先去缓存读取数据，如果缓存 MISS（数据没有被缓存），则需要从存储中读取数据，并将数据更新到缓存系统中，整个流程和代码如下所示：



## 缓存更新

我们把常见的缓存更新方案总结为两大类，业务层更新和外部组件更新，比较常见的是通过业务更新的方案。

### 业务层更新缓存

刚开始接触缓存方案的同学可能会纠结几个点，先更新缓存还是先更新存储，缓存的处理是通过删除来实现还是通过更新来实现。这里我们面临的问题本质上是一个数据库的分布式事务的问题，需要处理数据可靠性的挑战，并发更新带来的隔离性挑战，和数据更新原子性的挑战。

**数据可靠性**

如果要保证数据的可靠性，在业务逻辑成功之前，必须保障有一份数据落地，我们有以下两个选择：

先更新成功存储，再更新缓存；

先更新成功缓存，再跟新存储，如果存储更新失败，删除缓存；

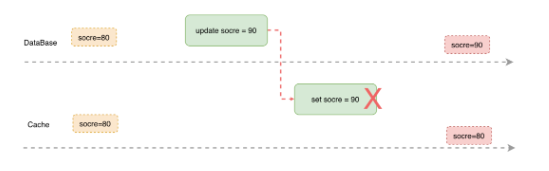
操作隔离性。

一条数据的更新涉及到存储和缓存两套系统，如果多个线程同时操作一条数据，并且没有方案保证多个操作之间的有序执行，就可能会发生更新顺序错乱导致数据不一致的问题。



**更新原子性**

引入缓存后，我们需要保证缓存和存储要么同时更新成功，要么同时更新失败，否则部分更新成功就会导致缓存和存储数据不一致的问题。



**业务层缓存更新方案**

我们看到大多数的常见是选择以下方案，保障数据可靠性，尽量减少数据不一致的出现，通过TTL超时机制在一定时间段后自动解决数据不一致现象。

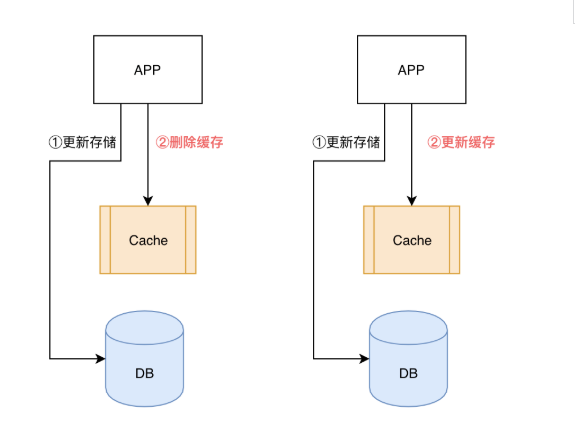
Step1：更新存储，保证数据可靠性；

Step2：更新缓存，2个策略怎么选：

惰性更新：删除缓存，等待下次读MISS再缓存（推荐方案）；

积极更新：将最新的值更新到缓存（不推荐）；

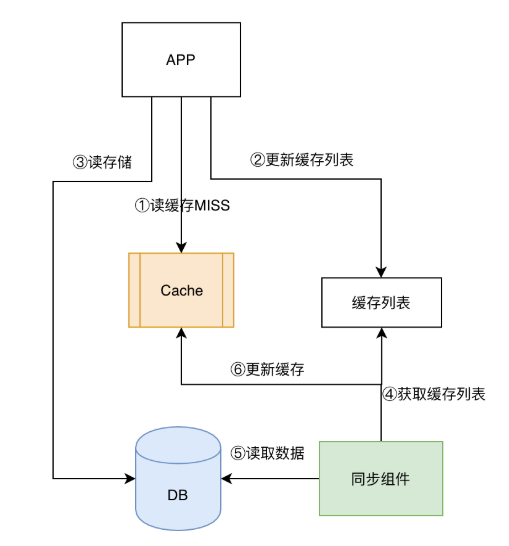
积极更新策略，缓存数据实时性更高，但是在缓存侧带来了更多的更新操作，这会提高更新冲突导致脏数据概率。



### 外部组件更新缓存

**缓存MISS处理方案**

在通过第三方组件更新的方案中，为了保障数据的一致性，避免对单条数据的并行更新，缓存的所有更新操作都需要交给同步组件，因此缓存 MISS 场景下的逻辑：



**缓存更新方案**

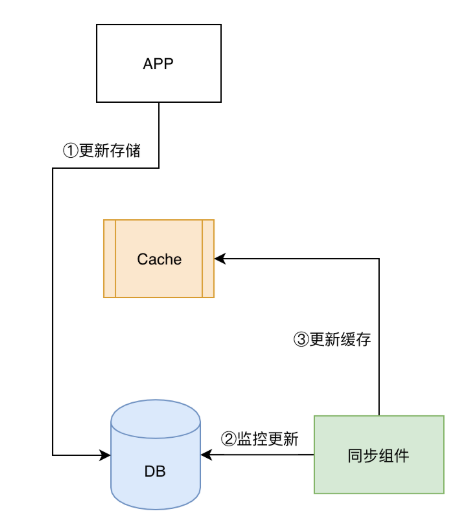
第一：需要监控存储的日志，或者通过Triger来监控存储数据的变更，需要对存储系统非常熟悉；

第二：需要对更新进行过滤，我们的目的是缓存热数据，但是像DDL、批量更新这一系列的操作是不需要更新缓存的，要把非业务更新操作过滤；

第三：同步组件需要理解数据，不通用；

先更新存储，由第三方组件异步更新缓存；

该方案投入较大，只适合特定的场景，并且有以下3个难点：

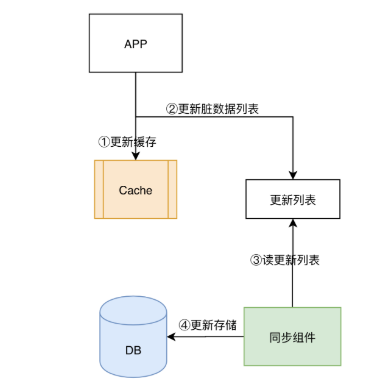


### 其他缓存更新方案

在实际的生产中，我们还会看到很多先更新缓存，然后通过第三方组件更新存储的场景，但是这个方案也会面临数据一致性和数据可靠性的挑战，虽然不推荐，但是确实还是能看到有在使用这个方案的，我们拿出来探讨下。

这个场景数据可靠性，不及先更新存储的方案，但是写入性能高，延迟低；

这个方案APP和第三方组件都会更新Cache，会存在数据一致性的问题，因为很难保障两个组件更新的时序。



## 缓存淘汰

缓存的作用是将热点数据缓存到内存实现加速，内存的成本要远高于磁盘，因此我们通常仅仅缓存热数据在内存，冷数据需要定期的从内存淘汰，数据的淘汰通常有两种方案：

**主动淘汰**，这是推荐的方式，我们通过对Key设置TTL的方式来让Key定期淘汰，以保障冷数据不会长久的占有内存。TTL的策略可以保证冷数据一定被淘汰，但是没有办法保障热数据始终在内存，这个我们在后面会展开；

**被动淘汰**，这个是保底方案，并不推荐，Redis提供了一系列的Maxmemory策略来对数据进行驱逐，触发的前提是内存要到达maxmemory（内存使用率 100%），在maxmemory的场景下缓存的质量是不可控的，因为每次缓存一个Key都可能需要去淘汰一个Key。

# 手写缓存

# 分布式数据库缓存方案