# 背景

由于数据库的缓存一般是针对查询的内容，而且粒度也比较小，一般只有表中的数据没有发生变动的时候，数据库的缓存才会产生作用。

但这并不能减少业务逻辑对数据库的增删改操作的 IO 压力，因此缓存技术应运而生，该技术实现了对热点数据的高速缓存，可以大大缓解后端数据库的压力。

# 概述

## 原理

Redis 的效率很高，官方给出的数据是 100000+QPS，这是因为：

1、Redis 完全基于内存，绝大部分请求是纯粹的内存操作，执行效率高。

2、Redis 使用单进程单线程模型的（K，V）数据库，将数据存储在内存中，存取均不会受到硬盘IO的限制，因此其执行速度极快。

另外单线程也能处理高并发请求，还可以避免频繁上下文切换和锁的竞争，如果想要多核运行也可以启动多个实例。

3、数据结构简单，对数据操作也简单，Redis不使用表，不会强制用户对各个关系进行关联，不会有复杂的关系限制，其存储结构就是键值对，类似于 HashMap，HashMap最大的优点就是存取的时间复杂度为O(1)。

4、Redis使用多路I/O复用模型，为非阻塞IO。

注：Redis采用的I/O多路复用函数：epoll/kqueue/evport/select。

**选用策略：**

1、因地制宜，优先选择时间复杂度为O(1)的I/O多路复用函数作为底层实现。

2、由于Select要遍历每一个IO，所以其时间复杂度为O(n)，通常被作为保底方案。

3、基于React设计模式监听I/O事件。

## Memcache

Memcache的代码层类似Hash，特点如下：

支持简单数据类型

不支持数据持久化存储

不支持主从

不支持分片

Redis 特点如下：

数据类型丰富

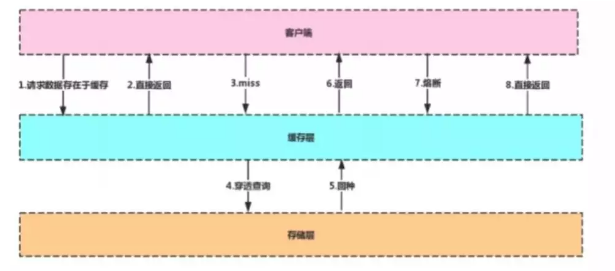
支持数据磁盘持久化存储

支持主从

支持分片

为什么 Redi

# 架构



客户端在对数据库发起请求时，先到缓存层查看是否有所需的数据，如果缓存层存有客户端所需的数据，则直接从缓存层返回，否则进行穿透查询，对数据库进行查询。

如果在数据库中查询到该数据，则将该数据回写到缓存层，以便下次客户端再次查询能够直接从缓存层获取数据。

# 数据类型

## String

最基本的数据类型，其值最大可存储 512M，二进制安全（Redis 的 String 可以包含任何二进制数据，包含 jpg 对象等）。

注：如果重复写入 key 相同的键值对，后写入的会将之前写入的覆盖。

## Hash

String 元素组成的字典，适用于存储对象。

## List

列表，按照 String 元素插入顺序排序。其顺序为后进先出。由于其具有栈的特性，所以可以实现如“最新消息排行榜”这类的功能。

## Set

String 元素组成的无序集合，通过哈希表实现（增删改查时间复杂度为 O(1)），不允许重复。

另外，当我们使用 Smembers 遍历 Set 中的元素时，其顺序也是不确定的，是通过 Hash 运算过后的结果。

Redis 还对集合提供了求交集、并集、差集等操作，可以实现如同共同关注，共同好友等功能。

## Sorted Set

通过分数来为集合中的成员进行从小到大的排序。

更高级的Redis类型

用于计数的 HyperLogLog、用于支持存储地理位置信息的 Geo。

# 缓存

## 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。

## 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

#### 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。

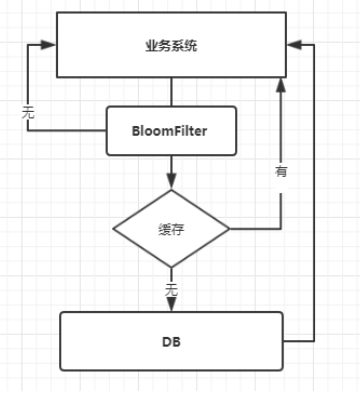
那么我们就可以为这些key 设置的值设置为null 丢到缓存里面去。后面再出现查询这个key 的请求的时候，直接返回null ,就不用在到 数据库中去走一圈了。但是别忘了设置过期时间。

#### 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。



### 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。

### 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

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。

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；

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。

## 缓存雪崩

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。

### 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。**

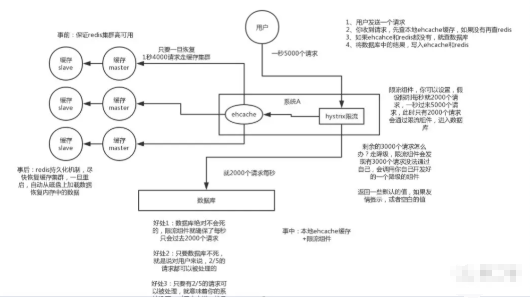
使用ehcache本地缓存的目的也是考虑在Redis Cluster完全不可用的时候ehcache本地缓存还能够支撑一阵。

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：



## 热点数据集中失效

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。

### 问题

### 解决

#### 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

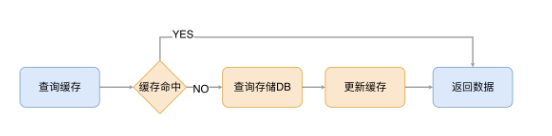
结合上面的击穿的情况，在第一个请求去查询数据库的时候对他加一个互斥锁，其余的查询请求都会被阻塞住，直到锁被释放，从而保护数据库。但是也是由于它会阻塞其他的线程，此时系统吞吐量会下降。需要结合实际的业务去考虑是否要这么做。

## 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；

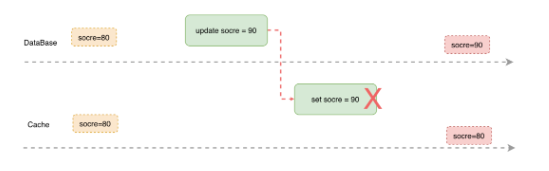
操作隔离性。

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。

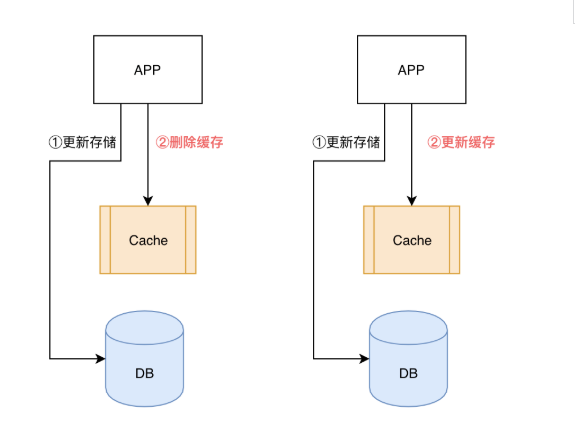
Step1：更新存储，保证数据可靠性；

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）；

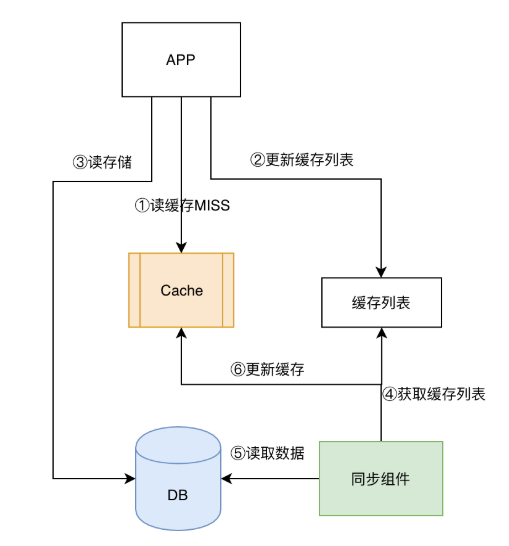
积极更新策略，缓存数据实时性更高，但是在缓存侧带来了更多的更新操作，这会提高更新冲突导致脏数据概率。



#### 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**

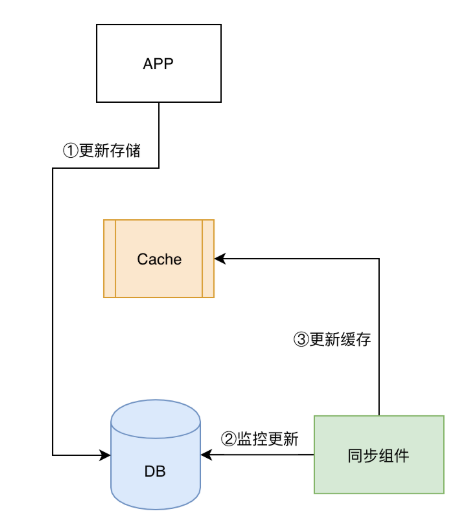
第一：需要监控存储的日志，或者通过 Triger 来监控存储数据的变更，需要对存储系统非常熟悉；

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：

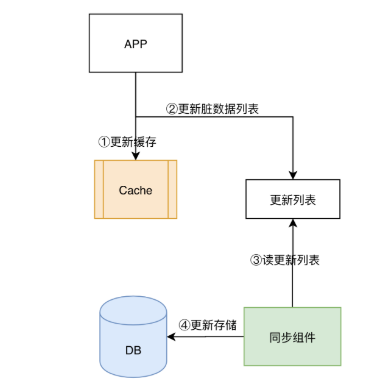


#### 其他缓存更新方案

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。

# 内存

## 内存大小设置

Redis是基于内存的key-value数据库，因为系统的内存大小有限，所以我们在使用Redis的时候可以配置Redis能使用的最大的内存大小。

### 通过配置文件配置

通过在Redis安装目录下面的redis.conf配置文件中添加以下配置设置内存大小：

//设置Redis最大占用内存大小为100M

maxmemory 100mb

redis的配置文件不一定使用的是安装目录下面的redis.conf文件，启动redis服务的时候是可以传一个参数指定redis的配置文件的。

### 通过命令修改

Redis支持运行时通过命令动态修改内存大小

//设置Redis最大占用内存大小为100M

127.0.0.1:6379> config set maxmemory 100mb

//获取设置的Redis能使用的最大内存大小

127.0.0.1:6379> config get maxmemory

如果不设置最大内存大小或者设置最大内存大小为0，在64位操作系统下不限制内存大小，在32位操作系统下最多使用3GB内存。

## 内存淘汰

既然可以设置Redis最大占用内存大小，那么配置的内存就有用完的时候。那在内存用完的时候，还继续往Redis里面添加数据不就没内存可用了吗？

实际上Redis定义了几种策略用来处理这种情况：

noeviction(默认策略) ：对于写请求不再提供服务，直接返回错误（DEL请求和部分特殊请求除外）

allkeys-lru：从所有key中使用LRU算法进行淘汰

volatile-lru：从设置了过期时间的key中使用LRU算法进行淘汰

allkeys-random：从所有key中随机淘汰数据

volatile-random：从设置了过期时间的key中随机淘汰

volatile-ttl：在设置了过期时间的key中，根据key的过期时间进行淘汰，越早过期的越优先被淘汰

当使用 volatile-lru、volatile-random、volatile-ttl

这三种策略时，如果没有key可以被淘汰，则和noeviction一样返回错误。

### 获取内存淘汰策略

获取当前内存淘汰策略：

127.0.0.1:6379> config get maxmemory-policy

通过配置文件设置淘汰策略（修改redis.conf文件）：

maxmemory-policy allkeys-lru

通过命令修改淘汰策略：

127.0.0.1:6379> config set maxmemory-policy allkeys-lru

### LRU算法

LRU(Least Recently Used)，即最近最少使用，是一种缓存置换算法。在使用内存作为缓存的时候，缓存的大小一般是固定的。当缓存被占满，这个时候继续往缓存里面添加数据，就需要淘汰一部分老的数据，释放内存空间用来存储新的数据。这个时候就可以使用LRU算法了。

其核心思想是：如果一个数据在最近一段时间没有被用到，那么将来被使用到的可能性也很小，所以就可以被淘汰掉。

**LRU在Redis中的实现**

1、近似LRU算法

Redis使用的是近似LRU算法，它跟常规的LRU算法还不太一样。近似LRU算法通过随机采样法淘汰数据，每次随机出5（默认）个key，从里面淘汰掉最近最少使用的key（MySQL是在5/7位置）。

可以通过maxmemory-samples参数修改采样数量：例：maxmemory-samples 10 maxmenory-samples配置的越大，淘汰的结果越接近于严格的LRU算法。

Redis为了实现近似LRU算法，给每个key增加了一个额外增加了一个24bit的字段，用来存储该key最后一次被访问的时间。

2、Redis3.0对近似LRU的优化

Redis3.0对近似LRU算法进行了一些优化。新算法会维护一个候选池（大小为16），池中的数据根据访问时间进行排序，第一次随机选取的key都会放入池中，随后每次随机选取的key只有在访问时间小于池中最小的时间才会放入池中，直到候选池被放满。当放满后，如果有新的key需要放入，则将池中最后访问时间最大（最近被访问）的移除。

当需要淘汰的时候，则直接从池中选取最近访问时间最小（最久没被访问）的key淘汰掉就行。

### LFU算法

LFU算法是Redis4.0里面新加的一种淘汰策略。它的全称是Least Frequently Used，它的核心思想是根据key的最近被访问的**频率**进行淘汰，很少被访问的优先被淘汰，被访问的多的则被留下来。

**LFU算法能更好的表示一个key被访问的热度**。假如你使用的是LRU算法，一个key很久没有被访问到，只刚刚是偶尔被访问了一次，那么它就被认为是热点数据，不会被淘汰，而有些key将来是很有可能被访问到的则被淘汰了。如果使用LFU算法则不会出现这种情况，因为使用一次并不会使一个key成为热点数据。

LFU一共有两种策略：

volatile-lfu：在设置了过期时间的key中使用LFU算法淘汰key

allkeys-lfu：在所有的key中使用LFU算法淘汰数据

注：要注意的一点是这两种策略只能在Redis4.0及以上设置，如果在Redis4.0以下设置会报错。

# 持久化

持久化，即将数据持久存储，而不因断电或其他各种复杂外部环境影响数据的完整性。

由于Redis将数据存储在内存而不是磁盘中，所以内存一旦断电，Redis中存储的数据也随即消失，这往往是用户不期望的，所以Redis有持久化机制来保证数据的安全性。

**Redis 如何做持久化？**

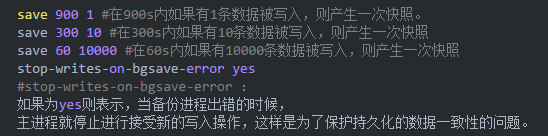
Redis目前有两种持久化方式，即**RDB和AOF**，RDB是通过保存某个时间点的**全量数据快照**实现数据的持久化，当恢复数据时，直接通过RDB文件中的快照，将数据恢复。

## RDB（快照）持久化

RDB持久化会在某个特定的间隔保存那个时间点的**全量数据的快照**。

### 原理

RDB 配置文件，redis.conf：



1、RDB 的创建与载入

**SAVE：**阻塞Redis的服务器进程，直到RDB文件被创建完毕。SAVE命令很少被使用，因为其会阻塞主线程来保证快照的写入，由于Redis是使用一个主线程来接收所有客户端请求，这样会阻塞所有客户端请求。

**BGSAVE：**该指令会Fork出一个子进程来创建RDB文件，不阻塞服务器进程，子进程接收请求并创建RDB快照，父进程继续接收客户端的请求。

子进程在完成文件的创建时会向父进程发送信号，父进程在接收客户端请求的过程中，在一定的时间间隔通过轮询来接收子进程的信号。

我们也可以通过使用lastsave指令来查看BGSAVE是否执行成功，lastsav可以返回最后一次执行成功BGSAVE的时间。

2、自动化触发RDB持久化的方式

自动化触发RDB持久化的方式如下：

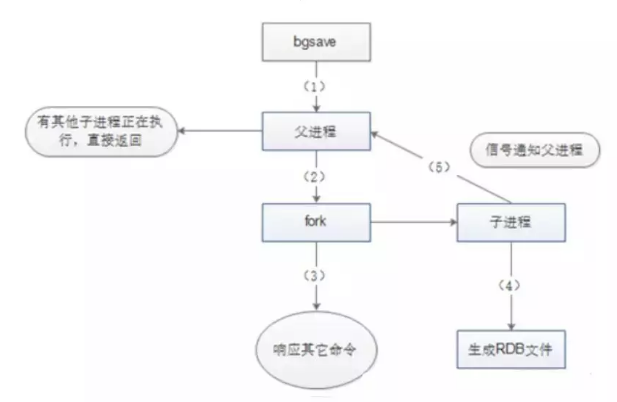
根据 redis.conf配置里的SAVE m n定时触发（实际上使用的是BGSAVE）。

主从复制时，主节点自动触发。

执行Debug Reload。

执行Shutdown且没有开启AOF持久化。

1. BGSAVE的原理



启动：

检查是否存在子进程正在执行AOF或者RDB的持久化任务。如果有则返回 false。

调用Redis源码中的rdbSaveBackground方法，方法中执行fork()产生子进程执行 RDB 操作。

关于fork()中的Copy-On-Write。

fork()在Linux中创建子进程采用Copy-On-Write（写时拷贝技术），即如果有多个调用者同时要求相同资源（如内存或磁盘上的数据存储）。

他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本给调用者，而其他调用者所见到的最初的资源仍然保持不变。

### 特点

RDB优点：全量数据快照，文件小，恢复快。

RDB缺点：无法保存最近一次快照之后的数据。

RDB持久化方式的缺点如下：

内存数据全量同步，数据量大的状况下，会由于I/O而严重影响性能。

可能会因为Redis宕机而丢失从当前至最近一次快照期间的数据。

## AOF持久化

AOF持久化（保存写状态）是通过**保存Redis的写状态**来记录数据库的。

相对RDB来说，RDB持久化是通过备份数据库的状态来记录数据库，而AOF持久化是备份数据库接收到的指令：

AOF记录除了查询以外的所有变更数据库状态的指令。

以增量的形式追加保存到 AOF 文件中。

### 开启AOF持久化

1、打开 redis.conf 配置文件，将appendonly属性改为yes。

2、修改 appendfsync属性，该属性可以接收三种参数，分别是always，everysec，no。

always表示总是即时将缓冲区内容写入AOF文件当中，everysec表示每隔一秒将缓冲区内容写入AOF文件，no表示将写入文件操作交由操作系统决定。

一般来说，操作系统考虑效率问题，会等待缓冲区被填满再将缓冲区数据写入 AOF 文件中。

appendonly yes

#appendsync always

appendfsync everysec

# appendfsync no

### 日志重写解决AOF文件不断增大

随着写操作的不断增加，AOF文件会越来越大。假设递增一个计数器100次，如果使用RDB持久化方式，我们只要保存最终结果100即可。

而AOF持久化方式需要记录下这100次递增操作的指令，而事实上要恢复这条记录，只需要执行一条命令就行，所以那一百条命令实际可以精简为一条。

Redis支持这样的功能，在不中断前台服务的情况下，可以重写AOF文件，同样使用到了COW（写时拷贝）。

**重写过程如下：**

调用 fork()，创建一个子进程。

子进程把新的 AOF 写到一个临时文件里，不依赖原来的 AOF 文件。

主进程持续将新的变动同时写到内存和原来的 AOF 里。

主进程获取子进程重写 AOF 的完成信号，往新 AOF 同步增量变动。

使用新的 AOF 文件替换掉旧的 AOF 文件。

### 特点

AOF优点：可读性高，适合保存增量数据，数据不易丢失。

AOF缺点：文件体积大，恢复时间长。

## RDB-AOF混合持久化方式

Redis 4.0之后推出了此种持久化方式，**RDB作为全量备份，AOF作为增量备份，并且将此种方式作为默认方式使用**。

在上述两种方式中，RDB方式是将全量数据写入RDB文件，这样写入的特点是文件小，恢复快，但无法保存最近一次快照之后的数据，AOF则将Redis指令存入文件中，这样又会造成文件体积大，恢复时间长等弱点。

在RDB-AOF方式下，持久化策略首先将缓存中数据以RDB方式全量写入文件，再将写入后新增的数据以AOF的方式追加在RDB数据的后面，在下一次做RDB持久化的时候将 AOF 的数据重新以RDB的形式写入文件。

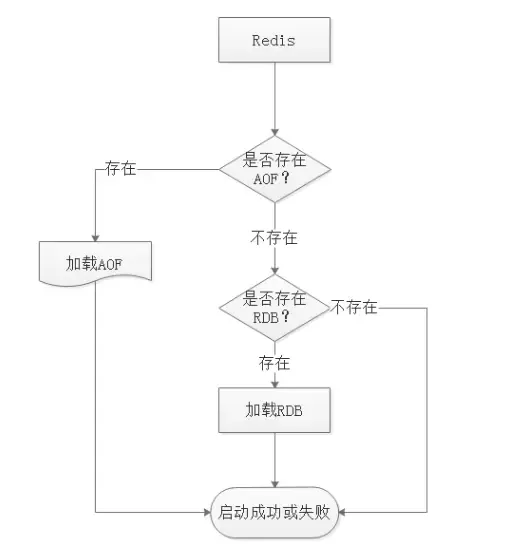
这种方式既可以提高读写和恢复效率，也可以减少文件大小，同时可以保证数据的完整性。

在此种策略的持久化过程中，子进程会通过管道从父进程读取增量数据，在以 RDB 格式保存全量数据时，也会通过管道读取数据，同时不会造成管道阻塞。

可以说，在此种方式下的持久化文件，前半段是RDB格式的全量数据，后半段是AOF格式的增量数据。此种方式是目前较为推荐的一种持久化方式。

# 恢复

RDB和AOF文件共存情况下的恢复流程如下图：



从图可知，Redis启动时会先检查AOF是否存在，如果AOF存在则直接加载AOF，如果不存在AOF，则直接加载RDB文件。

# Pipeline

Pipeline 和 Linux 的管道类似，它可以让 Redis 批量执行指令。

Redis 基于请求/响应模型，单个请求处理需要一一应答。如果需要同时执行大量命令，则每条命令都需要等待上一条命令执行完毕后才能继续执行，这中间不仅仅多了 RTT，还多次使用了系统 IO。

Pipeline 由于可以批量执行指令，所以可以节省多次 IO 和请求响应往返的时间。但是如果指令之间存在依赖关系，则建议分批发送指令。

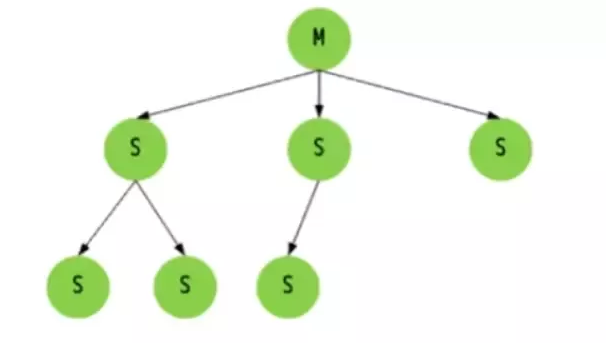
# 同步复制

## 主从同步

Redis一般是使用一个Master节点来进行写操作，而若干个Slave节点进行读操作，Master和Slave分别代表了一个个不同的Redis Server实例。

另外定期的数据备份操作也是单独选择一个Slave去完成，这样可以最大程度发挥Redis的性能，为的是保证数据的弱一致性和最终一致性。

另外，Master和Slave的数据不是一定要即时同步的，但是在一段时间后Master和Slave的数据是趋于同步的，这就是最终一致性。



**全同步过程如下：**

Slave 发送 Sync 命令到 Master。

Master 启动一个后台进程，将 Redis 中的数据快照保存到文件中。

Master 将保存数据快照期间接收到的写命令缓存起来。

Master 完成写文件操作后，将该文件发送给 Slave。

使用新的 AOF 文件替换掉旧的 AOF 文件。

Master 将这期间收集的增量写命令发送给 Slave 端。

**增量同步过程如下：**

Master 接收到用户的操作指令，判断是否需要传播到 Slave。

将操作记录追加到 AOF 文件。

将操作传播到其他 Slave：对齐主从库；往响应缓存写入指令。

将缓存中的数据发送给 Slave。

## Redis Sentinel（哨兵）

主从模式弊端：当 Master 宕机后，Redis 集群将不能对外提供写入操作。Redis Sentinel 可解决这一问题。

解决主从同步 Master 宕机后的主从切换问题：

监控：检查主从服务器是否运行正常。

提醒：通过 API 向管理员或者其它应用程序发送故障通知。

自动故障迁移：主从切换（在 Master 宕机后，将其中一个 Slave 转为 Master，其他的 Slave 从该节点同步数据）。

# 集群

如何从海量数据里快速找到所需？

1、分片

按照某种规则去划分数据，分散存储在多个节点上。通过将数据分到多个 Redis 服务器上，来减轻单个 Redis 服务器的压力。

2、一致性 Hash 算法

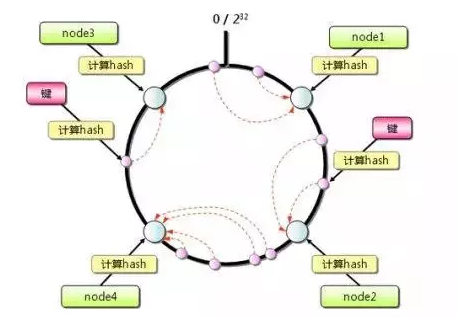
既然要将数据进行分片，那么通常的做法就是获取节点的 Hash 值，然后根据节点数求模。

但这样的方法有明显的弊端，当 Redis 节点数需要动态增加或减少的时候，会造成大量的 Key 无法被命中。所以 Redis 中引入了一致性 Hash 算法。

该算法对 2^32 取模，将 Hash 值空间组成虚拟的圆环，整个圆环按顺时针方向组织，每个节点依次为 0、1、2…2^32-1。

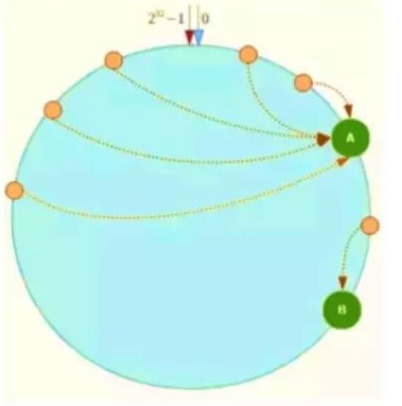
之后将每个服务器进行 Hash 运算，确定服务器在这个 Hash 环上的地址，确定了服务器地址后，对数据使用同样的 Hash 算法，将数据定位到特定的 Redis 服务器上。

如果定位到的地方没有 Redis 服务器实例，则继续顺时针寻找，找到的第一台服务器即该数据最终的服务器位置。



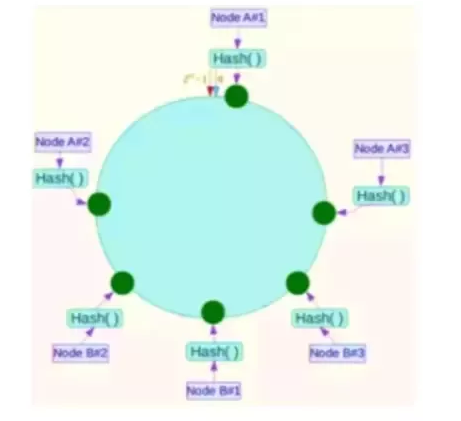
3、Hash 环的数据倾斜问题

Hash 环在服务器节点很少的时候，容易遇到服务器节点不均匀的问题，这会造成数据倾斜，数据倾斜指的是被缓存的对象大部分集中在 Redis 集群的其中一台或几台服务器上。



如上图，一致性 Hash 算法运算后的数据大部分被存放在 A 节点上，而 B 节点只存放了少量的数据，久而久之 A 节点将被撑爆。

针对这一问题，可以引入虚拟节点解决。简单地说，就是为每一个服务器节点计算多个 Hash，每个计算结果位置都放置一个此服务器节点，称为虚拟节点，可以在服务器 IP 或者主机名后放置一个编号实现。



例如上图：将 NodeA 和 NodeB 两个节点分为 Node A#1-A#3，NodeB#1-B#3。

# 应用

## 分布式锁

## 异步队列