

# 开篇寄语

---

## 开篇寄语：缓存，你真的用对了吗？

你好，我是你的缓存老师陈波，可能大家对我的网名 fishermen 会更熟悉。

我是资深老码农一枚，经历了新浪微博从起步到当前月活数亿用户的大型互联网系统的技术演进过程，现任新浪微博技术专家。我于 2008 年加入新浪，最初从事新浪 IM 的后端研发。2009 年之后开始微博 Feed 平台系统的研发及架构工作，深度参与最初若干个版本几乎所有业务的开发和架构改进，2013 年后开始从事微博平台基础架构相关的研发工作。目前主要从事微博 Feed 平台的基础设施、缓存中间件、分布式存储等的研发及架构优化工作。

那么，我们为什么要学习缓存呢？有必要学习缓存吗？

随着互联网从门户/搜索时代进入移动社交时代，互联网产品也从满足用户单向浏览的需求，发展为满足用户个性信息获取及社交的需求。这就要求产品做到以用户和关系为基础，对海量数据进行实时分析计算。也就意味着，用户的每次请求，服务后端都要查询用户的个人信息、社交关系图谱，以及关系图谱涉及到的大量关联信息。还要将这些信息进行聚合、过滤、筛选和排序，最终响应给用户。如果这些信息全部从 DB 中加载，将会是一个无法忍受的漫长等待过程。

而缓存的使用，是提升系统性能、改善用户体验的唯一解决之道。

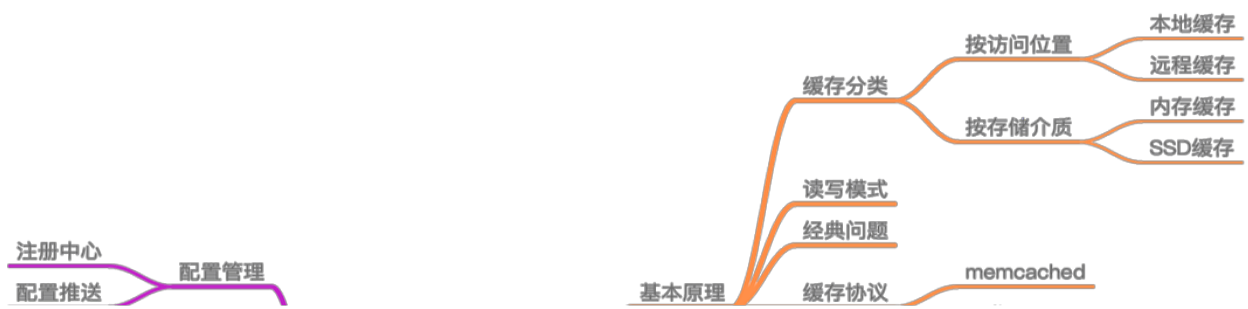
以新浪微博为例，作为移动互联网时代的一个开拓者和重量级社交分享平台，自 2009 年上线后，用户数量和微博数量都从 0 开启并高速增长，到 2019 年，日活跃用户已超 2 亿，每日新发 Feed 1~2 亿，每日访问量百亿级，历史数据高达千亿级。同时，在微博的日常服务中，核心接口可用性要达到 99.99%，响应时间在 10~60ms 以内，核心单个业务的数据访问量高达百万级 QPS。

所有这些数据都是靠良好的架构和不断改进的缓存体系来支撑的。

其实，作为互联网公司，只要有直接面对用户的业务，要想持续确保系统的访问性能和可用性，都需要使用缓存。因此，缓存也是后端工程师面试中一个非常重要的考察点，面试官通常会通过应聘者对缓存相关知识的理解深入程度，来判断其开发经验和学习能力。可以说，对缓存的掌握程度，在某种意义上决定了后端开发者的职业高度。

想学好缓存，需要掌握哪些知识呢？

可以看一下这张“缓存知识点全景图”。

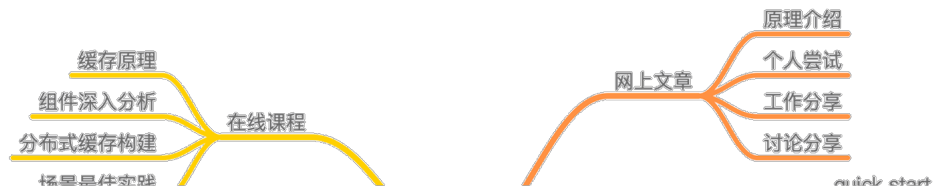


首先，要熟练掌握缓存的基础知识，了解缓存常用的分类、读写模式，熟悉缓存的七大经典问题及解决应对之策，同时要从缓存组件的访问协议、Client 入手，熟练掌握如何访问各种缓存组件，如 Memcached、Redis、Pika 等。

其次，要尽可能深入理解缓存组件的实现方案、设计原理，了解缓存的各种特性、优势和不足，这样在缓存数据与预期不一致时，能够快速定位并解决问题。

再次，还要多了解线上大中型系统是如何对缓存进行架构设计的。线上系统，业务功能丰富多变，跨域部署环境复杂，而且热点频发，用户习惯迥异。因此，缓存系统在设计之初就要尽量进行良好设计，规划好如何进行Hash及分布、如何保障数据的一致性、如何进行扩容和缩容。当然，缓存体系也需要伴随业务发展持续演进，这就需要对缓存体系进行持续的状态监控、异常报警、故障演练，以确保在故障发生时能及时进行人或自动化运维处理，并根据线上状况不断进行优化和改进。

最后，了解缓存在各种场景下的最佳实践，理解这些最佳实践背后的 Tradeoff，做到知其然知其所以然，以便在实际工作中能举一反三，把知识和经验更好的应用到工作实践中来。



如何高效学习缓存呢？你能学到什么？

对于缓存，网上学习资料很多，但过于零散和重复，想要系统地学习还是需要通过阅读缓存相关的书籍、论文和缓存源码，或是学习一些来自实战总结的网络课程。但前面几种形式目前都需要花费较多时间。为了学员既系统又快速地获得所需知识，拉勾教育推出了“300 分钟学会”系列技术课，其中“缓存”课由我来讲。

在这 300 分钟里，我将结合自己在微博平台的缓存架构经验，用 10 课时来分享：

如何更好地引入和使用缓存，自系统设计之初，就把缓存设计的关键点对号入座。

如何规避并解决缓存设计中的七大经典问题。

从协议、使用技巧、网络模型、核心数据结构、存储架构、数据处理模型、优化及改进方案等，多角度全方位深入剖析互联网企业大量使用的Memcached、Redis等开源缓存组件。

教你如何利用它们构建一个分布式缓存服务体系。

最后，我将结合诸如秒杀、海量计数、微博 Feed 聚合等经典业务场景，分析如何构建相应的高可用、高性能、易扩展的缓存架构体系。

通过本课程，你可以：

系统地学习缓存之设计架构的关键知识点；

学会如何更好地使用 Memcached、Redis 等缓存组件；

对这些缓存组件的内部架构、设计原理有一个较为深入的了解，真正做到知其然更知其所以然；

学会如何根据业务需要对缓存组件进行二次开发；

搞懂如何构建一个大型的分布式缓存服务系统；

了解在当前多种热门场景下缓存服务的最佳实践；

现学现用，针对互联网大中型系统，构建出一个更好的缓存架构体系，在大幅提升系统吞吐和响应性能的同时，达到高可用、高扩展，从而可以更从容地应对海量并发请求和极端热点事件。

## 《300分钟搞定分布式缓存》课程大纲

### 开篇寄语：缓存，你真的用对了吗？

第一章： 缓存的原理、引入及设计	1 业务数据访问性能太低怎么办？
	2 如何根据业务来选择缓存模式和组件？
	3 设计缓存架构时需要考量哪些因素？
第二章： 7 大缓存经典问题	4 缓存失效、穿透和雪崩问题怎么处理？
	5 缓存数据不一致和并发竞争怎么处理？
	6 Hot Key 和 Big Key 引发的问题怎么应对？
第三章： Memcached 的原理 及架构剖析	7 MC 为何是应用最广泛的缓存组件？
	8 MC 系统架构是如何布局的？
	9 MC 是如何使用多线程和状态机来处理请求命令的？
第四章：	10 MC 是怎么定位 key 的？
	11 MC 如何淘汰冷 key 和失效 key？

Memcached 进阶	12 为何 MC 能长期维持高性能读写？
	13 如何完整学习 MC 协议及优化 client 访问？
第五章： 分布式 Memcached 实战	14 大数据时代，MC 如何应对新的常见问题？
	15 如何深入理解、应用及扩展 Twemproxy？
	16 常用的缓存组件 Redis 是如何运行的？

# 第一章：缓存的原理、引入及设计

## 第01讲：业务数据访问性能太低怎么办？

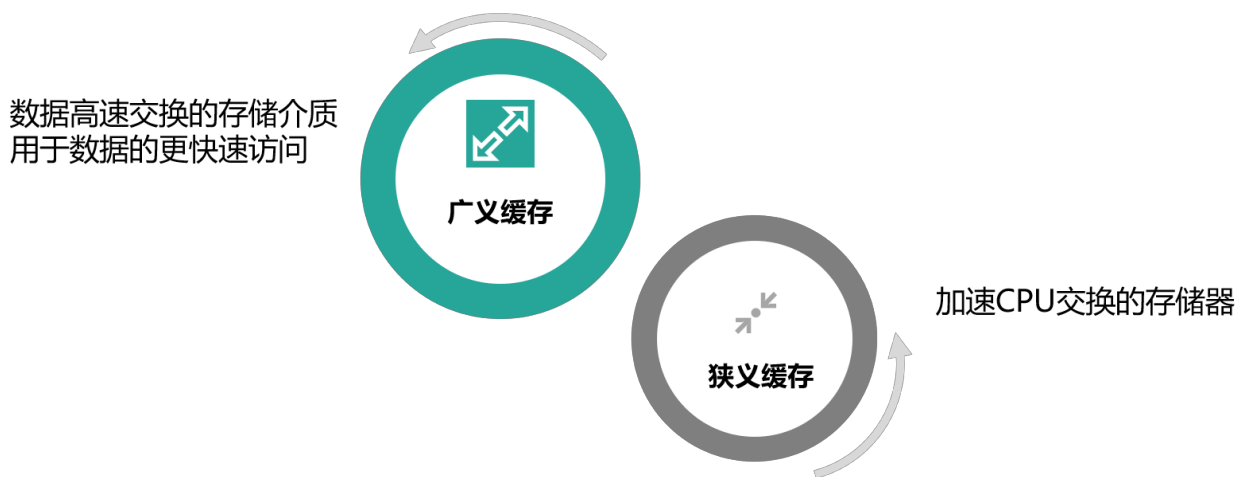
你好，我是你的缓存老师陈波，欢迎进入第1课时“缓存的原理”。这节课主要讲缓存的基本思想、缓存的优点、缓存的代价三个部分。

缓存的定义

先来看下缓存的定义。

缓存最初的含义，是指用于加速 CPU 数据交换的 RAM，即随机存取存储器，通常这种存储器使用更昂贵但快速的静态 RAM（SRAM）技术，用以对 DRAM 进行加速。这是一个狭义缓存的定义。

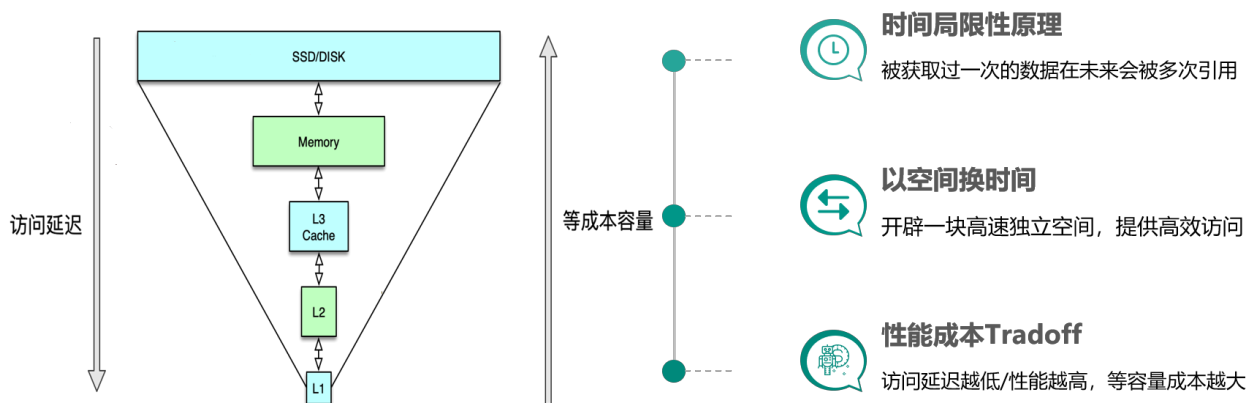
而广义缓存的定义则更宽泛，任何可以用于数据高速交换的存储介质都是缓存，可以是硬件也可以是软件。



缓存存在的意义就是通过开辟一个新的数据交换缓冲区，来解决原始数据获取代价太大的问题，让数据得到更快的访问。本课主要聚焦于广义缓存，特别是互联网产品大量使用的各种缓存组件和技术。

缓存原理

缓存的基本思想



缓存构建的基本思想是利用时间局限性原理，通过空间换时间来达到加速数据获取的目的，同时由于缓存空间的成本较高，在实际设计架构中还要考虑访问延迟和成本的权衡问题。这里面有 3 个关键点。

一是时间局限性原理，即被获取过一次的数据在未来会被多次引用，比如一条微博被一个人感兴趣并阅读后，它大概率还会被更多人阅读，当然如果变成热门微博后，会被数以百万/千万计算的更多用户查看。

二是以空间换时间，因为原始数据获取太慢，所以我们开辟一块高速独立空间，提供高效访问，来达到数据获取加速的目的。

三是性能成本 Tradeoff，构建系统时希望系统的访问性能越高越好，访问延迟越低小越好。但维持相同数据规模的存储及访问，性能越高延迟越小，成本也会越高，所以在系统架构设计时，你需要在系统性能和开发运行成本之间做取舍。比如左边这张图，相同成本的容量，SSD 硬盘容量会比内存大 10~30 倍以上，但读写延迟却高 50~100 倍。

## 缓存的优势

缓存的优势主要有以下几点：

提升访问性能

降低网络拥堵

减轻服务负载

增强可扩展性

通过前面的介绍，我们已经知道缓存存储原始数据，可以大幅提升访问性能。不过在实际业务场景中，缓存中存储的往往是需要频繁访问的中间数据甚至最终结果，这些数据相比 DB 中的原始数据小很多，这样就可以减少网络流量，降低网络拥堵，同时由于减少了解析和计算，调用方和存储服务的负载也可以大幅降低。缓存的读写性能很高，预热快，在数据访问存在性能瓶颈或遇到突发流量，系统读写压力大增时，可以快速部署上线，同时在流量稳定后，也可以随时下线，从而使系统的可扩展性大大增强。

### 缓存的代价

然而不幸的是，任何事情都有两面性，缓存也不例外，我们在享受缓存带来一系列好处的同时，也注定需要付出一定的代价。

首先，服务系统中引入缓存，会增加系统的复杂度。

其次，由于缓存相比原始 DB 存储的成本更高，所以系统部署及运行的费用也会更高。

最后，由于一份数据同时存在缓存和 DB 中，甚至缓存内部也会有多个数据副本，多份数据就会存在一致性问题，同时缓存体系本身也会存在可用性问题和分区的问题。这就需要我们加强对缓存原理、缓存组件以及优秀缓存体系实践的理解，从系统架构之初就对缓存进行良好设计，降低缓存引入的副作用，让缓存体系成为服务系统高效稳定运行的强力基石。

一般来讲，服务系统的全量原始数据存储在 DB 中（如 MySQL、HBase 等），所有数据的读写都可以通过 DB 操作来获取。但 DB 读写性能低、延迟高，如 MySQL 单实例的读写 QPS 通常只有千级别（3000~6000），读写平均耗时 10~100ms 级别，如果一个用户请求需要查 20 个不同的数据来聚合，仅仅 DB 请求就需要数百毫秒甚至数秒。而 cache 的读写性能正好可以弥补 DB 的不足，比如 Memcached 的读写 QPS 可以达到 10~100万 级别，读写平均耗时在 1ms 以下，结合并发访问技术，单个请求即便查上百条数据，也可以轻松应对。

但 cache 容量小，只能存储部分访问频繁的热数据，同时，同一份数据可能同时存在 cache 和 DB，如果处理不当，就会出现数据不一致的问题。所以服务系统在处理业务请求时，需要对 cache 的读写方式进行适当设计，既要保证数据高效返回，又要尽量避免数据不一致等各种问题。

好了，第 1 课时的内容到这里就全部结束了，我们一起来做一个简单的回顾。首先，这一课时，你先了解了缓存的定义以及基本思想。然后，又学习了缓存的优点和代价。

## 第02讲：如何根据业务来选择缓存模式和组件？

你好，我是你的缓存老师陈波，欢迎进入第 2 课时“缓存的读写模式及分类”。这一课时我们主要学习缓存的读写模式以及缓存的分类。

缓存读写模式

如下图，业务系统读写缓存有 3 种模式：

Cache Aside（旁路缓存）

Read/Write Through（读写穿透）

Write Behind Caching（异步缓存写入）



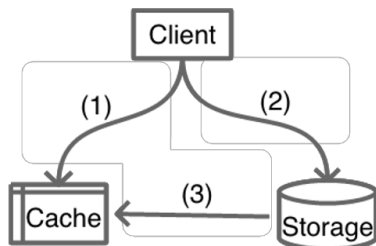
## 业务系统读写缓存3种模式

Cache Aside  
(旁路缓存)

Read/Write Through  
(读写穿透)

Write Behind Caching  
(异步缓存写入)

### Cache Aside



### Cache Aside

1

Write: 更新db, 删除cache, db驱动cache更新

2

Read: miss后读db+回写

3

特点: Lazy计算, 以DB数据为准

4

适合场景: 更强一致性 Cache数据构建复杂

如上图所示, Cache Aside 模式中, 业务应用方对于写, 是更新 DB 后, 直接将 key 从 cache 中删除, 然后由 DB 驱动缓存数据的更新; 而对于读, 是先读 cache, 如果 cache 没有, 则读 DB, 同时将从 DB 中读取的数据回写到 cache。

这种模式的特点是, 业务端处理所有数据访问细节, 同时利用 Lazy 计算的思想, 更新 DB 后, 直接删除 cache 并通过 DB 更新, 确保数据以 DB 结果为准, 则可以大幅降低 cache 和 DB 中数据不一致的概率。

如果没有专门的存储服务, 同时是对数据一致性要求比较高的业务, 或者是缓存数据更新比较复杂的业务, 这些情况都比较适合使用 Cache Aside 模式。如微博发展初期, 不少业务采用这种模式, 这些缓存数据需要通过多个原始数据进行计算后设置。在部分数据变更后, 直接删除缓存。同时, 使用一个 Trigger 组件, 实时读取 DB 的变更日志, 然后重新计算并更新缓存。如果读缓存的时候, Trigger 还没写入 cache, 则由调用方自行到 DB 加载计算并写入 cache。

### Read/Write Through

### Read/Write Through

Write: cache不存在更新DB; cache存在更新cache+DB

1

Read: miss后由缓存服务加载并写cache

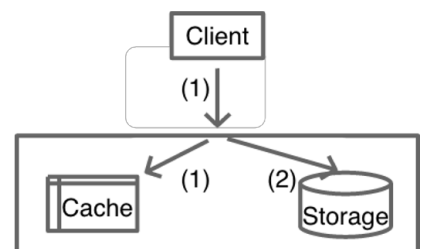
2

特点: 存储服务负责数据读写, 隔离型更佳, 热数据友好

3

适合场景: 数据有冷热区分

4



如上图，对于 Cache Aside 模式，业务应用需要同时维护 cache 和 DB 两个数据存储方，过于繁琐，于是就有了 Read/Write Through 模式。在这种模式下，业务应用只关注一个存储服务即可，业务方的读写 cache 和 DB 的操作，都由存储服务代理。存储服务收到业务应用的写请求时，会首先查 cache，如果数据在 cache 中不存在，则只更新 DB，如果数据在 cache 中存在，则先更新 cache，然后更新 DB。而存储服务收到读请求时，如果命中 cache 直接返回，否则先从 DB 加载，回种到 cache 后返回响应。

这种模式的特点是，存储服务封装了所有的数据处理细节，业务应用端代码只用关注业务逻辑本身，系统的隔离性更佳。另外，进行写操作时，如果 cache 中没有数据则不更新，有缓存数据才更新，内存效率更高。

微博 Feed 的 Outbox Vector（即用户最新微博列表）就采用这种模式。一些粉丝较少且不活跃的用户发表微博后，Vector 服务会首先查询 Vector Cache，如果 cache 中没有该用户的 Outbox 记录，则不写该用户的 cache 数据，直接更新 DB 后就返回，只有 cache 中存在才会通过 CAS 指令进行更新。

Write Behind Caching



Write Behind Caching 模式与 Read/Write Through 模式类似，也由数据存储服务来管理 cache 和 DB 的读写。不同点是，数据更新时，Read/write Through 是同步更新 cache 和 DB，而 Write Behind Caching 则是只更新缓存，不直接更新 DB，而是改为异步批量的方式来更新 DB。该模式的特点是，数据存储的写性能最高，非常适合一些变更特别频繁的业务，特别是可以合并写请求的业务，比如对一些计数业务，一条 Feed 被点赞 1 万次，如果更新 1 万次 DB 代价很大，而合并成一次请求直接加 1 万，则是一个非常轻量的操作。但这种模型有个显著的缺点，即数据的一致性变差，甚至在一些极端场景下可能会丢失数据。比如系统 Crash、机器宕机时，如果有数据还没保存到 DB，则会存在丢失的风险。所以这种读写模式适合变更频率特别高，但对一致性要求不太高的业务，这样写操作可以异步批量写入 DB，减小 DB 压力。

讲到这里，缓存的三种读写模式讲完了，你可以看到三种模式各有优劣，不存在最佳模式。实际上，我们也不可能设计出一个最佳的完美模式出来，如同前面讲到的空间换时间、访问延迟换低成本一样，高性能和强一致性从来都是有冲突的，系统设计从来就是取舍，随处需要 trade-off。这个思想会贯穿整个 cache 课程，这也许是我们学习这个课程的另外一个收获，即如何根据业务场景，更好的做 trade-off，从而设计出更好的服务系统。

缓存分类及常用缓存介绍

前面介绍了缓存的基本思想、优势、代价以及读写模式，接下来一起看下互联网企业常用的缓存有哪些分类。

#### 按宿主层次分类

按宿主层次分类的话，缓存一般可以分为本地 Cache、进程间 Cache 和远程 Cache。

本地 Cache 是指业务进程内的缓存，这类缓存由于在业务系统进程内，所以读写性能超高且无任何网络开销，但不足是会随着业务系统重启而丢失。

进程间 Cache 是本地独立运行的缓存，这类缓存读写性能较高，不会随着业务系统重启丢数据，并且可以大幅减少网络开销，但不足是业务系统和缓存都在相同宿主机，运维复杂，且存在资源竞争。

远程 Cache 是指跨机器部署的缓存，这类缓存因为独立设备部署，容量大且易扩展，在互联网企业使用最广泛。不过远程缓存需要跨机访问，在高读写压力下，带宽容易成为瓶颈。

本地 Cache 的缓存组件有 Ehcache、Guava Cache 等，开发者自己也可以用 Map、Set 等轻松构建一个自己专用的本地 Cache。进程间 Cache 和远程 Cache 的缓存组件相同，只是部署位置的差异罢了，这类缓存组件有 Memcached、Redis、Pika 等。

#### 按存储介质分类

还有一种常见的分类方式是按存储介质来分，这样可以分为内存型缓存和持久化型缓存。

内存型缓存将数据存储在内存在，读写性能很高，但缓存系统重启或 Crash 后，内存数据会丢失。

持久化型缓存将数据存储到 SSD/Fusion-IO 硬盘中，相同成本下，这种缓存的容量会比内存型缓存大 1 个数量级以上，而且数据会持久化落地，重启不丢失，但读写性能相对低 1~2 个数量级。Memcached 是典型的内存型缓存，而 Pika 以及其他基于 RocksDB 开发的缓存组件等则属于持久化型缓存。

## 第03讲：设计缓存架构时需要考量哪些因素？

你好，我是你的缓存老师陈波，欢迎进入第 3 课时“缓存的引入及架构设计”。

至此，缓存原理相关的主要知识点就讲完了，接下来会讲到如何引入缓存并进行设计架构，以及在缓存设计架构中的一些关键考量点。

#### 缓存的引入及架构设计

##### 缓存组件选择

在设计架构缓存时，你首先要选定缓存组件，比如要用 Local-Cache，还是 Redis、Memcached、Pika 等开源缓存组件，如果业务缓存需求比较特殊，你还要考虑是直接定制开发一个新的缓存组件，还是对开源缓存进行二次开发，来满足业务需要。

##### 缓存数据结构设计

确定好缓存组件后，你还要根据业务访问的特点，进行缓存数据结构的设计。对于直接简单 KV 读写的业务，你可以将这些业务数据封装为 String、Json、Protocol Buffer 等格式，序列化成字节序列，然后直接写入缓存中。读取时，先从缓存组件获取到数据的字节序列，再进行反序列化操作即可。对于只需要存取部分字段或需要在缓存端进行计算的业务，你可以把数据设计为 Hash、Set、List、Geo 等结

构，存储到支持复杂集合数据类型的缓存中，如 Redis、Pika 等。

## 缓存分布设计

确定了缓存组件，设计好了缓存数据结构，接下来就要设计缓存的分布。可以从 3 个维度来进行缓存分布设计。

首先，要选择分布式算法，是采用取模还是一致性 Hash 进行分布。取模分布的方案简单，每个 key 只会存在确定的缓存节点，一致性 Hash 分布的方案相对复杂，一个 key 对应的缓存节点不确定。但一致性 Hash 分布，可以在部分缓存节点异常时，将失效节点的数据访问均衡分散到其他正常存活的节点，从而更好地保证了缓存系统的稳定性。

其次，分布读写访问如何进行实施，是由缓存 Client 直接进行 Hash 分布定位读写，还是通过 Proxy 代理来进行读写路由？Client 直接读写，读写性能最佳，但需要 Client 感知分布策略。在缓存部署发生在线变化时，也需要及时通知所有缓存 Client，避免读写异常，另外，Client 实现也较复杂。而通过 Proxy 路由，Client 只需直接访问 Proxy，分布逻辑及部署变更都由 Proxy 来处理，对业务应用开发最友好，但业务访问多一跳，访问性能会有一定的损失。

最后，缓存系统运行过程中，如果待缓存的数据量增长过快，会导致大量缓存数据被剔除，缓存命中率会下降，数据访问性能会随之降低，这样就需要将数据从缓存节点进行动态拆分，把部分数据水平迁移到其他缓存节点。这个迁移过程需要考虑，是由 Proxy 进行迁移还是缓存 Server 自身进行迁移，甚至根本就不支持迁移。对于 Memcached，一般不支持迁移，对 Redis，社区版本是依靠缓存 Server 进行迁移，而对 Codis 则是通过 Admin、Proxy 配合后端缓存组件进行迁移。

## 缓存架构部署及运维管理

设计完毕缓存的分布策略后，接下来就要考虑缓存的架构部署及运维管理了。架构部署主要考虑如何对缓存进行分池、分层、分 IDC，以及是否需要进行异构处理。

核心的、高并发访问的不同数据，需要分别分拆到独立的缓存池中，进行分别访问，避免相互影响；访问量较小、非核心的业务数据，则可以混存。

对海量数据、访问超过 10~100 万 级的业务数据，要考虑分层访问，并且要分摊访问量，避免缓存过载。

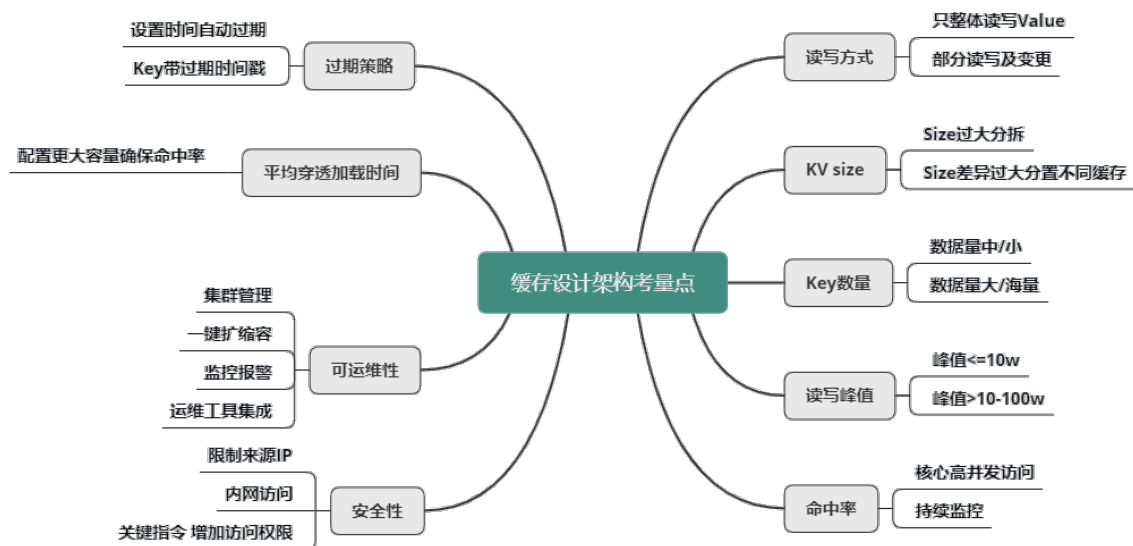
如果业务系统需要多 IDC 部署甚至异地多活，则需要对缓存体系也进行多 IDC 部署，要考虑如何跨 IDC 对缓存数据进行更新，可以采用直接跨 IDC 读写，也可以采用 DataBus 配合队列机进行不同 IDC 的消息同步，然后由消息处理机进行缓存更新，还可以由各个 IDC 的 DB Trigger 进行缓存更新。

某些极端场景下，还需要把多种缓存组件进行组合使用，通过缓存异构达到最佳读写性能。

站在系统层面，要想更好得管理缓存，还要考虑缓存的服务化，考虑缓存体系如何更好得进行集群管理、监控运维等。

## 缓存设计架构的常见考量点

在缓存设计架构的过程中，有一些非常重要的考量点，如下图所示，只有分析清楚了这些考量点，才能设计架构出更佳的缓存体系。



## 读写方式

首先是 value 的读写方式。是全部整体读写，还是只部分读写及变更？是否需要内部计算？比如，用户粉丝数，很多普通用户的粉丝有几千到几万，而大 V 的粉丝更是高达几千万甚至过亿，因此，获取粉丝列表肯定不能采用整体读写的方式，只能部分获取。另外在判断某用户是否关注了另外一个用户时，也不需要拉取该用户的全部关注列表，直接在关注列表上进行检查判断，然后返回 True/False 或 0/1 的方式更为高效。

## KV size

然后是不同业务数据缓存 KV 的 size。如果单个业务的 KV size 过大，需要分拆成多个 KV 来缓存。但是，不同缓存数据的 KV size 如果差异过大，也不能缓存在一起，避免缓存效率的低下和相互影响。

## key 的数量

key 的数量也是一个重要考虑因素。如果 key 数量不大，可以在缓存中存下全量数据，把缓存当 DB 存储来用，如果缓存读取 miss，则表明数据不存在，根本不需要再去 DB 查询。如果数据量巨大，则在缓存中尽可能只保留频繁访问的热数据，对于冷数据直接访问 DB。

## 读写峰值

另外，对缓存数据的读写峰值，如果小于 10 万级别，简单分拆到独立 Cache 池即可。而一旦数据的读写峰值超过 10 万 甚至到达 100 万 级的 QPS，则需要对 Cache 进行分层处理，可以同时使用 Local-Cache 配合远程 cache，甚至远程缓存内部继续分层叠加分池进行处理。微博业务中，大多数核心业务的 Memcached 访问都采用的这种处理方式。

## 命中率

缓存的命中率对整个服务体系的性能影响甚大。对于核心高并发访问的业务，需要预留足够的容量，确保核心业务缓存维持较高的命中率。比如微博中的 Feed Vector Cache，常年的命中率高达 99.5% 以上。为了持续保持缓存的命中率，缓存体系需要持续监控，及时进行故障处理或故障转移。同时在部分缓存节点异常、命中率下降时，故障转移方案，需要考虑是采用一致性 Hash 分布的访问漂移策略，还是采用数据多层备份策略。

## 过期策略



可以设置较短的过期时间，让冷 key 自动过期；

也可以让 key 带上时间戳，同时设置较长的过期时间，比如很多业务系统内部有这样一些 key：  
key\_20190801。

### 平均缓存穿透加载时间

平均缓存穿透加载时间在某些业务场景下也很重要，对于一些缓存穿透后，加载时间特别长或者需要复杂计算的数据，而且访问量还比较大的业务数据，要配置更多容量，维持更高的命中率，从而减少穿透到 DB 的概率，来确保整个系统的访问性能。

### 缓存可运维性

对于缓存的可运维性考虑，则需要考虑缓存体系的集群管理，如何进行一键扩缩容，如何进行缓存组件的升级和变更，如何快速发现并定位问题，如何持续监控报警，最好有一个完善的运维平台，将各种运维工具进行集成。

### 缓存安全性

对于缓存的安全性考虑，一方面可以限制来源 IP，只允许内网访问，同时对于一些关键性指令，需要增加访问权限，避免被攻击或误操作时，导致严重后果。

好了，第3课时的内容到这里就全部结束了，我们一起来做一个简单的回顾。首先，我们学习了在系统研发中，如何引入缓存，如何按照4步走对缓存进行设计架构及管理。最后，还熟悉了缓存设计架构中的考量点，这样你在缓存设计架构时对号入座即可。

## 第二章：7大缓存经典问题

### 第04讲：缓存失效、穿透和雪崩问题怎么处理？

你好，我是你的缓存老师陈波，欢迎进入第 4 课时“缓存访问相关的经典问题”。

前面讲解了缓存的原理、引入，以及设计架构，总结了缓存在使用及设计架构过程中的很多套路和关键考量点。实际上，在缓存系统的设计架构中，还有很多坑，很多的明枪暗箭，如果设计不当会导致很多严重的后果。设计不当，轻则请求变慢、性能降低，重则会数据不一致、系统可用性降低，甚至会导致缓存雪崩，整个系统无法对外提供服务。

接下来将对缓存设计中的 7 大经典问题，如下图，进行问题描述、原因分析，并给出日常研发中，可能会出现该问题的业务场景，最后给出这些经典问题的解决方案。本课时首先学习缓存失效、缓存穿透与缓存雪崩。

## 缓存失效

### 问题描述

缓存第一个经典问题是缓存失效。上一课时讲到，服务系统查数据，首先会查缓存，如果缓存数据不存在，就进一步查 DB，最后查到数据后回种到缓存并返回。缓存的性能比 DB 高 50~100 倍以上，所以我们希望数据查询尽可能命中缓存，这样系统负荷最小，性能最佳。缓存里的数据存储基本上都是以 key 为索引进行存储和获取的。业务访问时，如果大量的 key 同时过期，很多缓存数据访问都会 miss，进而穿透到 DB，DB 的压力就会明显上升，由于 DB 的性能较差，只在缓存的 1%~2% 以下，这样请求的慢查率会明显上升。这就是缓存失效的问题。

### 原因分析

导致缓存失效，特别是很多 key 一起失效的原因，跟我们日常写缓存的过期时间息息相关。

在写缓存时，我们一般会根据业务的访问特点，给每种业务数据预置一个过期时间，在写缓存时把这个过期时间带上，让缓存数据在这个固定的过期时间后被淘汰。一般情况下，因为缓存数据是逐步写入的，所以也是逐步过期被淘汰的。但在某些场景，一大批数据会被系统主动或被动从 DB 批量加载，然后写入缓存。这些数据写入缓存时，由于使用相同的过期时间，在经历这个过期时间之后，这批数据就会一起到期，从而被缓存淘汰。此时，对这批数据的所有请求，都会出现缓存失效，从而都穿透到 DB，DB 由于查询量太大，就很容易压力大增，请求变慢。

### 业务场景

很多业务场景，稍不注意，就出现大量的缓存失效，进而导致系统 DB 压力大、请求变慢的情况。比如同一批火车票、飞机票，当可以售卖时，系统会一次性加载到缓存，如果缓存写入时，过期时间按照预先设置的过期值，那过期时间到期后，系统就会因缓存失效出现变慢的问题。类似的业务场景还有很多，比如微博业务，会有后台离线系统，持续计算热门微博，每当计算结束，会将这批热门微博批量写入对应的缓存。还比如，很多业务，在部署新 IDC 或新业务上线时，会进行缓存预热，也会一次性加载大批热数据。

### 解决方案

对于批量 key 缓存失效的问题，原因既然是预置的固定过期时间，那解决方案也从这里入手。设计缓存的过期时间时，使用公式：过期时间=base 时间+随机时间。即相同业务数据写缓存时，在基础过期时间之上，再加一个随机的过期时间，让数据在未来一段时间内慢慢过期，避免瞬时全部过期，对 DB 造成过大压力，如下图所示。

## 缓存穿透

### 问题描述

第二个经典问题是缓存穿透。缓存穿透是一个很有意思的问题。因为缓存穿透发生的概率很低，所以一般很难被发现。但是，一旦你发现了，而且量还不小，你可能立即就会经历一个忙碌的夜晚。因为对于正常访问，访问的数据即便不在缓存，也可以通过 DB 加载回种到缓存。而缓存穿透，则意味着有特殊访客在查询一个不存在的 key，导致每次查询都会穿透到 DB，如果这个特殊访客再控制一批肉鸡机器，持续访问你系统里不存在的 key，就会对 DB 产生很大的压力，从而影响正常服务。

### 原因分析

缓存穿透存在的原因，就是因为我们在系统设计时，更多考虑的是正常访问路径，对特殊访问路径、异常访问路径考虑相对欠缺。



缓存访问设计的正常路径，是先访问 cache，cache miss 后查 DB，DB 查询到结果后，回种缓存返回。这对于正常的 key 访问是没有问题的，但是如果用户访问的是一个不存在的 key，查 DB 返回空（即一个 NULL），那就不会把这个空写回cache。那以后不管查询多少次这个不存在的 key，都会 cache miss，都会查询 DB。整个系统就会退化成一个“前端+DB”的系统，由于 DB 的吞吐只在 cache 的 1%~2% 以下，如果有特殊访客，大量访问这些不存在的 key，就会导致系统的性能严重退化，影响正常用户的访问。

#### 业务场景

缓存穿透的业务场景很多，比如通过不存在的 UID 访问用户，通过不存在的车次 ID 查看购票信息。用户输入错误，偶尔几个这种请求问题不大，但如果是大量这种请求，就会对系统影响非常大。

#### 解决方案

那么如何解决这种问题呢？如下图所示。

第一种方案就是，查询这些不存在的数据时，第一次查 DB，虽然没查到结果返回 NULL，仍然记录这个 key 到缓存，只是这个 key 对应的 value 是一个特殊设置的值。

第二种方案是，构建一个 BloomFilter 缓存过滤器，记录全量数据，这样访问数据时，可以直接通过 BloomFilter 判断这个 key 是否存在，如果不存在直接返回即可，根本无需查缓存和 DB。

不过这两种方案在设计时仍然有一些要注意的坑。

对于方案一，如果特殊访客持续访问大量的不存在的 key，这些 key 即便只存一个简单的默认值，也会占用大量的缓存空间，导致正常 key 的命中率下降。所以进一步的改进措施是，对这些不存在的 key 只存较短的时间，让它们尽快过期；或者将这些不存在的 key 存在一个独立的公共缓存，从缓存查找时，先查正常的缓存组件，如果 miss，则查一下公共的非法 key 的缓存，如果后者命中，直接返回，否则穿透 DB，如果查出来是空，则回种到非法 key 缓存，否则回种到正常缓存。

对于方案二，BloomFilter 要缓存全量的 key，这就要求全量的 key 数量不大，10亿 条数据以内最佳，因为 10亿 条数据大概要占用 1.2GB 的内存。也可以用 BloomFilter 缓存非法 key，每次发现一个 key 是不存在的非法 key，就记录到 BloomFilter 中，这种记录方案，会导致 BloomFilter 存储的 key 持续高速增长，为了避免记录 key 太多而导致误判率增大，需要定期清零处理。

## BloomFilter

BloomFilter 是一个非常有意思的数据结构，不仅仅可以挡住非法 key 攻击，还可以低成本、高性能地对海量数据进行判断，比如一个系统有数亿用户和百亿级新闻 feed，就可以用 BloomFilter 来判断某个用户是否阅读某条新闻 feed。下面来对 BloomFilter 数据结构做一个分析，如下图所示。

BloomFilter 的目的是检测一个元素是否存在于一个集合内。它的原理，是用 bit 数据组来表示一个集合，对一个 key 进行多次不同的 Hash 检测，如果所有 Hash 对应的 bit 位都是 1，则表明 key 非常大概率存在，平均单记录占用 1.2 字节即可达到 99%，只要有一次 Hash 对应的 bit 位是 0，就说明这个 key 肯定不存在于这个集合内。

BloomFilter 的算法是，首先分配一块内存空间做 bit 数组，数组的 bit 位初始值全部设为 0，加入元素时，采用 k 个相互独立的 Hash 函数计算，然后将元素 Hash 映射的 K 个位置全部设置为 1。检测 key 时，仍然用这 k 个 Hash 函数计算出 k 个位置，如果位置全部为 1，则表明 key 存在，否则不存在。

BloomFilter 的优势是，全内存操作，性能很高。另外空间效率非常高，要达到 1% 的误判率，平均单条记录占用 1.2 字节即可。而且，平均单条记录每增加 0.6 字节，还可让误判率继续变为之前的 1/10，即平均单条记录占用 1.8 字节，误判率可以达到 1/1000；平均单条记录占用 2.4 字节，误判率可以到 1/10000，以此类推。这里的误判率是指，BloomFilter 判断某个 key 存在，但它实际不存在的概率，因为它存的是 key 的 Hash 值，而非 key 的值，所以有概率存在这样的 key，它们内容不同，但多次 Hash 后的 Hash 值都相同。对于 BloomFilter 判断不存在的 key，则是 100% 不存在的，反证法，如果这个 key 存在，那它每次 Hash 后对应的 Hash 值位置肯定是 1，而不会是 0。

缓存雪崩

问题描述

第三个经典问题是缓存雪崩。系统运行过程中，缓存雪崩是一个非常严重的问题。缓存雪崩是指部分缓存节点不可用，导致整个缓存体系甚至甚至服务系统不可用的情况。缓存雪崩按照缓存是否 rehash（即是否漂移）分两种情况：

缓存不支持 rehash 导致的系统雪崩不可用

缓存支持 rehash 导致的缓存雪崩不可用

原因分析

在上述两种情况中，缓存不进行 rehash 时产生的雪崩，一般是由于较多缓存节点不可用，请求穿透导致 DB 也过载不可用，最终整个系统雪崩不可用的。而缓存支持 rehash 时产生的雪崩，则大多跟流量洪峰有关，流量洪峰到达，引发部分缓存节点过载 Crash，然后因 rehash 扩散到其他缓存节点，最终整个缓存体系异常。

第一种情况比较容易理解，如下图所示。缓存节点不支持 rehash，较多缓存节点不可用时，大量 Cache 访问会失败，根据缓存读写模型，这些请求会进一步访问 DB，而且 DB 可承载的访问量要远比缓存小的多，请求量过大，就很容易造成 DB 过载，大量慢查询，最终阻塞甚至 Crash，从而导致服务异常。

第二种情况是怎么回事呢？这是因为缓存分布设计时，很多同学会选择一致性 Hash 分布方式，同时在部分节点异常时，采用 rehash 策略，即把异常节点请求平均分散到其他缓存节点。在一般情况下，一致性 Hash 分布+rehash 策略可以很好得运行，但在较大的流量洪峰来临之时，如果大流量 key 比较集中，正好在某 1~2 个缓存节点，很容易将这些缓存节点的内存、网卡过载，缓存节点异常 Crash，然后这些异常节点下线，这些大流量 key 请求又被 rehash 到其他缓存节点，进而导致其他缓存节点也被过载 Crash，缓存异常持续扩散，最终导致整个缓存体系异常，无法对外提供服务。

业务场景

缓存雪崩的业务场景并不少见，微博、Twitter 等系统在运行的最初若干年都遇到过很多次。比如，微博最初很多业务缓存采用一致性 Hash+rehash 策略，在突发洪水流量来临时，部分缓存节点过载 Crash 甚至宕机，然后这些异常节点的请求转到其他缓存节点，又导致其他缓存节点过载异常，最终整个缓存池过载。另外，机架断电，导致业务缓存多个节点宕机，大量请求直接打到 DB，也导致 DB 过载而阻塞，整个系统异常。最后缓存机器复电后，DB 重启，数据逐步加热后，系统才逐步恢复正常。

解决方案

预防缓存雪崩，这里给出 3 个解决方案。

方案一，对业务 DB 的访问增加读写开关，当发现 DB 请求变慢、阻塞，慢请求超过阈值时，就会关闭读开关，部分或所有读 DB 的请求进行 failfast 立即返回，待 DB 恢复后再打开读开关，如下图。

方案二，对缓存增加多个副本，缓存异常或请求 miss 后，再读取其他缓存副本，而且多个缓存副本尽量部署在不同机架，从而确保在任何情况下，缓存系统都会正常对外提供服务。

方案三，对缓存体系进行实时监控，当请求访问的慢速比超过阈值时，及时报警，通过机器替换、服务替换进行及时恢复；也可以通过各种自动故障转移策略，自动关闭异常接口、停止边缘服务、停止部分非核心功能措施，确保在极端场景下，核心功能的正常运行。

实际上，微博平台系统，这三种方案都采用了，通过三管齐下，规避缓存雪崩的发生。

## 第05讲：缓存数据不一致和并发竞争怎么处理？

你好，我是你的缓存老师陈波，欢迎进入第5课时“缓存数据相关的经典问题”。

## 数据不一致

### 问题描述

七大缓存经典问题的第四个问题是数据不一致。同一份数据，可能会同时存在 DB 和缓存之中。那就有可能发生，DB 和缓存的数据不一致。如果缓存有多个副本，多个缓存副本里的数据也可能发生不一致现象。

### 原因分析

不一致的问题大多跟缓存更新异常有关。比如更新 DB 后，写缓存失败，从而导致缓存中存的是老数据。另外，如果系统采用一致性 Hash 分布，同时采用 rehash 自动漂移策略，在节点多次上下线之后，也会产生脏数据。缓存有多个副本时，更新某个副本失败，也会导致这个副本的数据是老数据。

### 业务场景

导致数据不一致的场景也不少。如下图所示，在缓存机器的带宽被打满，或者机房网络出现波动时，缓存更新失败，新数据没有写入缓存，就会导致缓存和 DB 的数据不一致。缓存 rehash 时，某个缓存机器反复异常，多次上下线，更新请求多次 rehash。这样，一份数据存在多个节点，且每次 rehash 只更新某个节点，导致一些缓存节点产生脏数据。

### 解决方案

要尽量保证数据的一致性。这里也给出了 3 个方案，可以根据实际情况进行选择。

第一个方案，cache 更新失败后，可以进行重试，如果重试失败，则将失败的 key 写入队列机服务，待缓存访问恢复后，将这些 key 从缓存删除。这些 key 在再次被查询时，重新从 DB 加载，从而保证数据的一致性。

第二个方案，缓存时间适当调短，让缓存数据及早过期后，然后从 DB 重新加载，确保数据的最终一致性。

第三个方案，不采用 rehash 漂移策略，而采用缓存分层策略，尽量避免脏数据产生。

## 数据并发竞争

### 问题描述

第五个经典问题是数据并发竞争。互联网系统，线上流量较大，缓存访问中很容易出现数据并发竞争的现象。数据并发竞争，是指在高并发访问场景，一旦缓存访问没有找到数据，大量请求就会并发查询 DB，导致 DB 压力大增的现象。

数据并发竞争，主要是由于多个进程/线程中，有大量并发请求获取相同的数据，而这个数据 key 因为正好过期、被剔除等各种原因在缓存中不存在，这些进程/线程之间没有任何协调，然后一起并发查询 DB，请求那个相同的 key，最终导致 DB 压力大增，如下图。

### 业务场景

数据并发竞争在大流量系统也比较常见，比如车票系统，如果某个火车车次缓存信息过期，但仍然有大量用户在查询该车次信息。又比如微博系统中，如果某条微博正好被缓存淘汰，但这条微博仍然有大量的转发、评论、赞。上述情况都会造成该车次信息、该条微博存在并发竞争读取的问题。

### 解决方案

要解决并发竞争，有 2 种方案。

方案一是使用全局锁。如下图所示，即当缓存请求 miss 后，先尝试加全局锁，只有加全局锁成功的线程，才可以到 DB 去加载数据。其他进程/线程在读取缓存数据 miss 时，如果发现这个 key 有全局锁，就进行等待，待之前的线程将数据从 DB 回种到缓存后，再从缓存获取。

方案二是，对缓存数据保持多个备份，即便其中一个备份中的数据过期或被剔除了，还可以访问其他备份，从而减少数据并发竞争的情况，如下图。

## 第06讲：Hot Key和Big Key引发的问题怎么应对？

你好，我是你的缓存老师陈波，欢迎进入第6课时“缓存特殊 key 相关的经典问题”。

Hot key

问题描述

第六个经典问题是 Hot key。对于大多数互联网系统，数据是分冷热的。比如最近的新闻、新发表的微博被访问的频率最高，而比较久远的之前的新闻、微博被访问的频率就会小很多。而在突发事件发生时，大量用户同时去访问这个突发热点信息，访问这个 Hot key，这个突发热点信息所在的缓存节点就容易出现过载和卡顿现象，甚至会被 Crash。

原因分析

Hot key 引发缓存系统异常，主要是因为突发热门事件发生时，超大量的请求访问热点事件对应的 key，比如微博中数十万、数百万的用户同时去吃一个新瓜。数十万的访问请求同一个 key，流量集中在一个缓存节点机器，这个缓存机器很容易被打到物理网卡、带宽、CPU 的极限，从而导致缓存访问变慢、卡顿。

业务场景

引发 Hot key 的业务场景很多，比如明星结婚、离婚、出轨这种特殊突发事件，比如奥运、春节这些重大活动或节日，还比如秒杀、双12、618 等线上促销活动，都很容易出现 Hot key 的情况。

解决方案

要解决这种极热 key 的问题，首先要找出这些 Hot key 来。对于重要节假日、线上促销活动、集中推送这些提前已知的事情，可以提前评估出可能的热 key 来。而对于突发事件，无法提前评估，可以通过 Spark，对应流任务进行实时分析，及时发现新发布的热点 key。而对于之前已发出的事情，逐步发酵成为热 key 的，则可以通过 Hadoop 对批处理任务离线计算，找出最近历史数据中的高频热 key。

找到热 key 后，就有很多解决办法了。首先可以将这些热 key 进行分散处理，比如一个热 key 名字叫 hotkey，可以被分散为 hotkey#1、hotkey#2、hotkey#3，.....hotkey#n，这 n 个 key 分散存在多个缓存节点，然后 client 端请求时，随机访问其中某个后缀的 hotkey，这样就可以把热 key 的请求打散，避免一个缓存节点过载，如下图所示。

其次，也可以 key 的名字不变，对缓存提前进行多副本+多级结合的缓存架构设计。

再次，如果热 key 较多，还可以通过监控体系对缓存的 SLA 实时监控，通过快速扩容来减少热 key 的冲击。

最后，业务端还可以使用本地缓存，将这些热 key 记录在本地缓存，来减少对远程缓存的冲击。

Big key

问题描述

最后一个经典问题是 Big key，也就是大 Key 的问题。大 key，是指在缓存访问时，部分 Key 的 Value 过大，读写、加载易超时的现象。

原因分析

造成这些大 key 慢查询的原因很多。如果这些大 key 占总体数据的比例很小，存 Mc，对应的 slab 较少，导致很容易被频繁剔除，DB 反复加载，从而导致查询较慢。如果业务中这种大 key 很多，而这种 key 被大量访问，缓存组件的网卡、带宽很容易被打满，也会导致较多的大 key 慢查询。另外，如果大 key 缓存的字段较多，每个字段的变更都会引发对这个缓存数据的变更，同时这些 key 也会被频繁地读取，读写相互影响，也会导致慢查现象。最后，大 key 一旦被缓存淘汰，DB 加载可能需要花费很多时间，这也会导致大 key 查询慢的问题。

业务场景

大 key 的业务场景也比较常见。比如互联网系统中需要保存用户最新 1 万个粉丝的业务，比如一个用户个人信息缓存，包括基本资料、关系图谱计数、发 feed 统计等。微博的 feed 内容缓存也很容易出现，一般用户微博在 140 字以内，但很多用户也会发表 1 千字甚至更长的微博内容，这些长微博也就成了大 key，如下图。

解决方案

对于大 key，给出 3 种解决方案。

第一种方案，如果数据存在 Mc 中，可以设计一个缓存阈值，当 value 的长度超过阈值，则对内容启用压缩，让 KV 尽量保持小的 size，其次评估大 key 所占的比例，在 Mc 启动之初，就立即预写足够数据的大 key，让 Mc 预先分配足够多的 trunk size 较大的 slab。确保后面系统运行时，大 key 有足够的空间来进行缓存。

第二种方案，如果数据存在 Redis 中，比如业务数据存 set 格式，大 key 对应的 set 结构有几千万个元素，这种写入 Redis 时会消耗很长的时间，导致 Redis 卡顿。此时，可以扩展新的数据结构，同时让 client 在这些大 key 写缓存之前，进行序列化构建，然后通过 restore 一次性写入，如下图所示。

第三种方案时，如下图所示，将大 key 分拆为多个 key，尽量减少大 key 的存在。同时由于大 key 一旦穿透到 DB，加载耗时很大，所以可以对这些大 key 进行特殊照顾，比如设置较长的过期时间，比如缓存内部在淘汰 key 时，同等条件下，尽量不淘汰这些大 key。

至此，本课时缓存的 7 大经典问题全部讲完。



我们要认识到，对于互联网系统，由于实际业务场景复杂，数据量、访问量巨大，需要提前规避缓存使用中的各种坑。你可以通过提前熟悉 Cache 的经典问题，提前构建防御措施，避免大量 key 同时失效，避免不存在 key 访问的穿透，减少大 key、热 key 的缓存失效，对热 key 进行分流。你可以采取一系列措施，让访问尽量命中缓存，同时保持数据的一致性。另外，你还可以结合业务模型，提前规划 cache 系统的 SLA，如 QPS、响应分布、平均耗时等，实施监控，以方便运维及时应对。在遇到部分节点异常，或者遇到突发流量、极端事件时，也能通过分池分层策略、key 分拆等策略，避免故障发生。

最终，你能在各种复杂场景下，面对高并发、海量访问，面对突发事件和洪峰流量，面对各种网络或机器硬件故障，都能保持服务的高性能和高可用。

## 第三章：Memcached的原理及架构剖析

---

### 第07讲：MC为何是应用最广泛的缓存组件？

你好，我是你的缓存老师陈波，欢迎你进入第 7 课时“Memcached 原理及特性”的学习。

众所周知，用户体验可以说是互联网企业最看重的指标，而在用户体验中，请求响应速度首当其冲。因此互联网系统对性能的追求是永无止境的。性能争霸，缓存为王，Memcached，作为互联网系统使用最广泛、影响最大的标配缓存组件，可以说的上是王中之王了。

本课时将讲解 Memcached 的原理及特性，系统架构，还会重点讲解 Memcached 的网络模型、状态机，最后还会涉及到 Memcached 命令处理的整个流程。

Memcached 原理及特性

首先来看 Memcached 的原理及特性。

原理

Memcached 是一个开源的、高性能的分布式 key/value 内存缓存系统。它以 key/value 键值对的方式存储数据，是一个键值类型的 NoSQL 组件。

NoSQL 即 Not SQL，泛指非关系型数据存储。NoSQL 是通过聚合模型来进行数据处理的。其聚合模型主要分为：key/value 键值对、列族、图形等几种方式。其中 key/value 键值类似我们平常使用的 map，只能通过 key 来进行查找和变更操作。我们使用的 Memcached、Redis 等都是 key/value 类型的 NoSQL 存储组件。

Memcached 简称 Mc，是一个典型的内存型缓存组件，这就意味着，Mc 一旦重启就会丢失所有的数据。如下图所示，Mc 组件之间相互不通信，完全由 client 对 key 进行 Hash 后分布和协同。Mc 采用多线程处理请求，由一个主线程和任意多个工作线程协作，从而充分利用多核，提升 IO 效率。

slab 机制

接下来介绍 Mc 的 slab 机制。

Mc 并不是将所有数据放在一起进行管理的，而是将内存划分为一系列相同大小的 slab 空间后，每个 slab 只管理一定范围内的数据存储。也就是说 Mc 内部采用 slab 机制来管理内存分配。Mc 内的内存分配以 slab 为单位，默认情况下一个 slab 是 1MB，可以通过 -l 参数在启动时指定其他数值。

slab 空间内部，会被进一步划分为一系列固定大小的 chunk。每个 chunk 内部存储一个 Item，利用 Item 结构存储数据。因为 chunk 大小固定，而 key/value 数据的大小随机。所以，Item 存储完 key/value 数据后，一般还会有多余的空间，这个多余的空间就被浪费了。为了提升内存的使用效率，chunk size 就不能太大，而要尽量选择与 key/value size 接近的，从而减少 chunk 内浪费的空间。

Mc 在分配内存时，先将内存按固定大小划分成 slab，然后再将不同 slab 分拆出固定 size 的 chunk。虽然 slab 内的 chunk 大小相同，但不同 slab 的 chunk size 并不同，Mc 会按照一个固定比例，使划分的 chunk size 逐步增大，从而满足不同大小 key/value 存储的需要。

如下图，一组具有相同 chunk size 的所有 slab，就组成一个 slabclass。不同 slabclass 的 chunk size 按递增因子一次增加。Mc 就通过 slabclass 来管理一组 slab 内的存储空间的。每个 slabclass 内部有一个 freelist，包含这组 slab 里所有空闲的 chunk，当需要存储数据时，从这个 freelist 里面快速分配一个 chunk 做存储空间。当 Item 数据淘汰剔除时，这个 Item 所在的 chunk 又被回收至这个 freelist。

Mc 在通过 slab 机制管理内存分配时，实际 key/value 是存在 Item 结构中，所以对 key/value 的存储空间分配就转换为对 Item 的分配。而 Item 空间的分配有 2 种方式，如果 Mc 有空闲空间，则从 slabclass 的 freelist 分配；如果没有空闲空间，则从对应 slabclass id 对应的 LRU 中剔除一个 Item，来复用这个 Item 的空间。

在查找或变更一个 key 时，首先要定位这个 key 所在的存储位置。Mc 是通过哈希表 Hashtable 来定位 key 的。Hashtable 可以看作是一个内存空间连续的大数组，而这个大数据的每一个槽位对应一个 key 的 Hash 值，这个槽位也称 bucket。由于不同 key 的 Hash 值可能相同，所以 Mc 在 Hashtable 的每个桶内部再用一个单向链表，来解决 Hash 冲突的问题。

Mc 内部是通过 LRU 来管理存储 Item 数据的，当内存不足时，会从 LRU 队尾中剔除一个过期或最不活跃的 key，供新的 Item 使用。

特性

讲完 slab 机制我们来学习 Mc 的特性。



Mc 最大的特性是高性能，单节点压测性能能达到百万级的 QPS。

其次因为 Mc 的访问协议很简单，只有 `get/set/cas/touch/gat/stats` 等有限的几个命令。Mc 的访问协议简单，跟它的存储结构也有关系。

Mc 存储结构很简单，只存储简单的 `key/value` 键值对，而且对 `value` 直接以二进制方式存储，不识别内部存储结构，所以有限几个指令就可以满足操作需要。

Mc 完全基于内存操作，在系统运行期间，在有新 `key` 写进来时，如果没有空闲内存分配，就会对最不活跃的 `key` 进行 `eviction` 剔除操作。

最后，Mc 服务节点运行也特别简单，不同 Mc 节点之间互不通信，由 `client` 自行负责管理数据分布。

## 第08讲：MC系统架构是如何布局的？

你好，我是你的缓存老师陈波，欢迎你进入第 8 课时“Memcached 系统架构”的学习。  
系统架构

我们来看一下 Mc 的系统架构。

如下图所示，Mc 的系统架构主要包括网络处理模块、多线程处理模块、哈希表、LRU、slab 内存分配模块 5 部分。Mc 基于 Libevent 实现了网络处理模块，通过多线程并发处理用户请求；基于哈希表对 `key` 进行快速定位，基于 LRU 来管理冷数据的剔除淘汰，基于 slab 机制进行快速的内存分配及存储。

### 系统架构

Mc 基于 Libevent 开发实现了多线程网络模型。Mc 的多线程网络模型分为主线程、工作线程。这些线程通过多路复用 IO 来进行网络 IO 接入以及读写处理。在 Linux 下，通常使用 `epoll`。通过多路复用 IO，特别是 `epoll` 的使用，Mc 线程无须遍历整个被侦听的描述符集，只要在被通知后遍历 Ready 队列的描述符集合就 OK 了。这些描述符是在各项准备工作完成之后，才被内核 IO 事件异步通知。也就是说，只在连接做好准备后，系统才会进行事件通知，Mc 才会进行 I/O 操作。这样就不会发生阻塞，使 Mc 在支持高并发的同时，拥有非常高的 IO 吞吐效率。

Mc 除了用于 IO 的主线程和工作线程外，还用于多个辅助线程，如 Item 爬虫线程、LRU 维护线程、哈希表维护线程等，通过多线程并发工作，Mc 可以充分利用机器的多个核心，实现很好的网络 IO 性能和数据处理能力。

Mc 通过哈希表即 Hashtable 来快速定位 `key`。数据存储时，数据 Item 结构在存入 slab 中的 chunk 后，也会被存放到 Hashtable 中。同时，Mc 的哈希表会在每个桶，通过 Item 记录一个单向链表，以此来解决不同 `key` 在哈希表中的 Hash 冲突问题。当需要查找给定 `key` 的 Item 时，首先计算 `key` 的 Hash 值，然后对哈希表中与 Hash 值对应的 bucket 中进行搜索，通过轮询 bucket 里的单向链表，找到该 `key` 对应的 Item 指针，这样就找到了 `key` 对应的存储 Item，如下图所示。

正常情况下，Mc 对哈希表的插入、查找操作都是在主表中进行的。当表中 Item 数量大于哈希表 bucket 节点数的 1.5 倍时，就对哈希表进行扩容。如下图所示，扩容时，Mc 内部使用两张 Hashtable，一个主哈希表 `primary_hashtable`，一个是旧哈希表 `old_hashtable`。当扩容开始时，原来的主哈希表就成为旧哈希表，而新分配一个 2 倍容量的哈希表作为新的主表。扩容过程中，维护线程会将旧表的 Item 指针，逐步复制插入到新主哈希表。迁移过程中，根据迁移位置，用户请求会同时查旧表和新的主表，当数据全部迁移完成，所有的操作就重新回到主表中进行。

## LRU 机制

Mc 主要通过 LRU 机制，来进行冷数据淘汰的。自 1.4.24 版本之后，Mc 不断优化 LRU 算法，当前 Mc 版本已默认启用分段 LRU 了。在启用分段 LRU 之前，每个 slabclass id 只对应一个 COLD LRU，在内存不足时，会直接从 COLD LRU 剔除数据。而在启用分段 LRU 之后，每个 slabclass id 就有 TEMP、HOT、WARM 和 COLD 四个 LRU。

如下图所示，TEMP LRU 中 Item 剩余过期时间通常很短，默认是 61 秒以内。该列队中的 Item 永远不会发生在队列内搬运，也不会迁移到其他队列。在插入新 key/value 时，如果 key 的剩余过期时间小于 61 秒，则直接进入 TEMP LRU。后面，在必要时直接进行过期即可。这样避免了锁竞争，性能也更高。

对于 HOT LRU，内部不搬运，当队列满时，如果队尾 Item 是 Active 状态，即被访问过，那么会迁移到 WARM 队列，否则迁移到 COLD 队列。

对于 WARM LRU，如果队列的 Item 被再次访问，就搬到队首，否则迁移到 COLD 队列。

对于 COLD LRU，存放的是最不活跃的 Item，一旦内存满了，队尾的 Item 会被剔除。如果 COLD LRU 里的 Item 被再次访问，会迁移到 WARM LRU。

## slab 分配机制

一般应用系统的内存分配是直接采用 `malloc` 和 `free` 来进行分配及回收的。长时间运行后，内存碎片越来越多，严重增加系统内存管理器的负担。碎片的不断产生，不仅导致大量的内存浪费，而且碎片整理越来越复杂，会导致内存分配越来越慢，进而导致系统分配速度和存储效率越来越差。Mc 的 slab 分配机制的出现，碎片问题迎刃而解。下面我们先简单了解一下 Mc 的 slab 分配机制。

Mc 通过 slab 机制来分配管理内存的，如下图所示。可以说，slab 分配机制的使用，是 Mc 分配及存储高性能的关键所在。在 Mc 启动时，会创建 64 个 slabclass，但索引为 0 的 slabclass 做 slab 重新分配之用，基本不参与其他 slabclass 的日常分配活动。每个 slabclass 会根据需要不断分配默认大小为 1MB 的 slab。

每个 slab 又被分为相同大小的 chunk。chunk 就是 Mc 存储数据的基本存储单位。slabclass 1 的 chunk size 最小，默认最小 chunk 的大小是 102 字节，后续的 slabclass 会按照增长因子逐步增大 chunk size，具体数值会进一步对 8 取整。Mc 默认的增长因子是 1.25，启动时可以通过 `-f` 将增长因子设为其他值。比如采用默认值，slabclass 1 的 chunk size 是 102，slabclass 2 的 chunk size 是  $102 \times 1.25$ ，再对 8 取整后是 128。

Mc slab 中的 chunk 中通过 Item 结构存 key/value 键值对，Item 结构体的头部存链表的指针、flag、过期时间等，然后存 key 及 value。一般情况下，Item 并不会将 chunk 填满，但由于每个 key/value 在存储时，都会根据 key/value size，选择最接近的 slabclass，所以 chunk 浪费的字节非常有限，基本可以忽略。

每次新分配一个 slab 后，会将 slab 空间等分成相同 size 的 chunk，这些 chunk 会被加入到 slabclass 的 freelist 中，在需要时进行分配。分配出去的 chunk 存储 Item 数据，在过期被剔除后，会再次进入 freelist，供后续使用。

## 第09讲：MC是如何使用多线程和状态机来处理请求命令的？

你好，我是你的缓存老师陈波，欢迎你进入第 9 课时“Memcached 网络模型及状态机”的学习。

### 网络模型

了解了 Mc 的系统架构之后，我们接下来可以逐一深入学习 Mc 的各个模块了。首先，我们来学习 Mc 的网络模型。

#### 主线程

Mc 基于 Libevent 实现多线程网络 IO 模型。Mc 的 IO 处理线程分主线程和工作线程，每个线程各有一个 event\_base，来监听网络事件。主线程负责监听及建立连接。工作线程负责对建立连接进行网络 IO 读取、命令解析、处理及响应。

Mc 主线程在监听端口时，当有连接到来，主线程 accept 该连接，并将连接调度给工作线程。调度处理逻辑，主线程先将 fd 封装成一个 CQ\_ITEM 结构，并存入新连接队列中，然后轮询一个工作线程，并通过管道向该工作线程发送通知。工作线程监听到通知后，会从新连接队列获取一个连接，然后开始从这个连接读取网络 IO 并处理，如下图所示。主线程的这个处理逻辑主要在状态机中执行，对应的连接状态为 conn\_listening。

#### 工作线程

工作线程监听到主线程的管道通知后，会从连接队列弹出一个新连接，然后就会创建一个 conn 结构体，注册该 conn 读事件，然后继续监听该连接上的 IO 事件。后续这个连接有命令进来时，工作线程会读取 client 发来的命令，进行解析并处理，最后返回响应。工作线程的主要处理逻辑也是在状态机中，一个名叫 drive\_machine 的函数。

#### 状态机

这个状态机由主线程和工作线程共享，实际是采用 switch-case 来实现的。状态机函数如下图所示，switch 连接的 state，然后根据连接的不同状态，执行不同的逻辑操作，并进行状态转换。接下来我们开始分析 Mc 的状态机。

#### 主线程状态机

如下图所示，主线程在状态机中只处理 `conn_listening` 状态，负责 `accept` 新连接和调度新连接给工作线程。状态机中其他状态处理基本都在工作线程中进行。由于 `Mc` 同时支持 `TCP`、`UDP` 协议，而互联网企业大多使用 `TCP` 协议，并且通过文本协议，来访问 `Mc`，所以后面状态机的介绍，将主要结合 `TCP` 文本协议来进行重点分析。

## 工作线程状态机

工作线程的状态机处理逻辑，如下图所示，包括刚建立 `conn` 连接结构体时进行的一些重置操作，然后注册读事件，在有数据进来时，读取网络数据，并进行解析并处理。如果是读取指令或统计指令，至此就基本处理完毕，接下来将响应写入连接缓冲。如果是更新指令，在进行初步处理后，还会继续读取 `value` 部分，再进行存储或变更，待变更完毕后将响应写入连接缓冲。最后再将响应写给 `client`。响应 `client` 后，连接会再次重置连接状态，等待进入下一次的命令处理循环中。这个过程主要包含了 `conn_new_cmd`、`conn_waiting`、`conn_read`、`conn_parse_cmd`、`conn_nread`、`conn_write`、`conn_mwrite`、`conn_closing` 这 8 个状态事件。

## 工作线程状态事件及逻辑处理

### `conn_new_cmd`

主线程通过调用 `dispatch_conn_new`，把新连接调度给工作线程后，`worker` 线程创建 `conn` 对象，这个连接初始状态就是 `conn_new_cmd`。除了通过新建连接进入 `conn_new_cmd` 状态之外，如果连接命令处理完毕，准备接受新指令时，也会将连接的状态设置为 `conn_new_cmd` 状态。

进入 `conn_new_cmd` 后，工作线程会调用 `reset_cmd_handler` 函数，重置 `conn` 的 `cmd` 和 `substate` 字段，并在必要时对连接 `buf` 进行收缩。因为连接在处理 `client` 来的命令时，对于写指令，需要分配较大的读 `buf` 来存待更新的 `key value`，而对于读指令，则需要分配较大的写 `buf` 来缓冲待发送给 `client` 的 `value` 结果。持续运行中，随着大 `size value` 的相关操作，这些缓冲会占用很多内存，所以需要设置一个阈值，超过阈值后就进行缓冲内存收缩，避免连接占用太多内存。在后端服务以及中间件开发中，这个操作很重要，因为线上服务的连接很容易达到万级别，如果一个连接占用几十 `KB` 以上的内存，后端系统仅连接就会占用数百 `MB` 甚至数 `GB` 以上的内存空间。

### `conn_parse_cmd`

工作线程处理完 `conn_new_cmd` 状态的主要逻辑后，如果读缓冲区有数据可以读取，则进入 `conn_parse_cmd` 状态，否则就会进入到 `conn_waiting` 状态，等待网络数据进来。

### `conn_waiting`

连接进入 `conn_waiting` 状态后，处理逻辑很简单，直接通过 `update_event` 函数注册读事件即可，之后会将连接状态更新为 `conn_read`。

### `conn_read`

当工作线程监听到网络数据进来，连接就进入 `conn_read` 状态。对 `conn_read` 的处理，是通过 `try_read_network` 从 `socket` 中读取网络数据。如果读取失败，则进入 `conn_closing` 状态，关闭连接。如果没有读取到任何数据，则会返回 `conn_waiting`，继续等待 `client` 端的数据到来。如果读取数据成功，则会将读取的数据存入 `conn` 的 `rbuf` 缓冲，并进入 `conn_parse_cmd` 状态，准备解析 `cmd`。

### `conn_parse_cmd`

`conn_parse_cmd` 状态的处理逻辑就是解析命令。工作线程首先通过 `try_read_command` 读取连接的读缓冲，并通过 `\n` 来分隔数据报文的命令。如果命令首行长度大于 1024，关闭连接，这就意味着 `key` 长度加上其他各项命令字段的总长度要小于 1024 字节。当然对于 `key`，`Mc` 有个默认的最大长度，`key_max_length`，默认设置为 250 字节。校验完毕首行报文的长度，接下来会在 `process_command`

函数中对首行指令进行处理。

process\_command 用来处理 Mc 的所有协议指令，所以这个函数非常重要。process\_command 会首先按照空格分拆报文，确定命令协议类型，分派给 process\_XX\_command 函数处理。

Mc 的命令协议从直观逻辑上可以分为获取类型、变更类型、其他类型。但从实际处理层面区分，则可以细分为 get 类型、update 类型、delete 类型、算术类型、touch 类型、stats 类型，以及其他类型。对应的处理函数为，process\_get\_command, process\_update\_command, process\_arithmetic\_command, process\_touch\_command 等。每个处理函数能够处理不同的协议，具体参见下图所示思维导图。

#### conn\_parse\_cmd

注意 conn\_parse\_cmd 的状态处理，只有读取到 \n，有了完整的命令首行协议，才会进入 process\_command，否则会跳转到 conn\_waiting，继续等待客户端的命令数据报文。在 process\_command 处理中，如果是获取类命令，在获取到 key 对应的 value 后，则跳转到 conn\_mwrite，准备写响应给连接缓冲。而对于 update 变更类型的指令，则需要继续读取 value 数据，此时连接会跳转到 conn\_nread 状态。在 conn\_parse\_cmd 处理过程中，如果遇到任何失败，都会跳转到 conn\_closing 关闭连接。

#### complete\_nread

对于 update 类型的协议指令，从 conn 继续读取 value 数据。读取到 value 数据后，会调用 complete\_nread，进行数据存储处理；数据处理完毕后，向 conn 的 wbuf 写响应结果。然后 update 类型处理的连接进入到 conn\_write 状态。

#### conn\_write

连接 conn\_write 状态处理逻辑很简单，直接进入 conn\_mwrite 状态。或者当 conn 的 iovused 为 0 或对于 udp 协议，将响应写入 conn 消息缓冲后，再进入 conn\_mwrite 状态。

#### conn\_mwrite

进入 conn\_mwrite 状态后，工作线程将通过 transmit 来向客户端写数据。如果写数据失败，跳转到 conn\_closing，关闭连接退出状态机。如果写数据成功，则跳转到 conn\_new\_cmd，准备下一次新指令的获取。

#### conn\_closing

最后一个 conn\_closing 状态，前面提到过很多次，在任何状态的处理过程中，如果出现异常，就会进入到这个状态，关闭连接，这个连接也就 Game Over 了。

#### Mc 命令处理全流程

至此，Mc 的系统架构和状态机的内容就全部讲完了，再梳理一遍 Mc 对命令的处理全过程，如下图所示，从而加深对 Mc 的状态机及命令处理流程的理解。

Mc 启动后，主线程监听并准备接受新连接接入。当有新连接接入时，主线程进入 conn\_listening 状态，accept 新连接，并将新连接调度给工作线程。

Worker 线程监听管道，当收到主线程通过管道发送的消息后，工作线程中的连接进入 `conn_new_cmd` 状态，创建 `conn` 结构体，并做一些初始化重置操作，然后进入 `conn_waiting` 状态，注册读事件，并等待网络 IO。

有数据到来时，连接进入 `conn_read` 状态，读取网络数据。

读取成功后，就进入 `conn_parse_cmd` 状态，然后根据 Mc 协议解析指令。

对于读取指令，获取到 `value` 结果后，进入 `conn_mwrite` 状态。

对于变更指令，则进入 `conn_nread`，进行 `value` 的读取，读取到 `value` 后，对 `key` 进行变更，当变更完毕后，进入 `conn_write`，然后将结果写入缓冲。然后和读取指令一样，也进入 `conn_mwrite` 状态。

进入到 `conn_mwrite` 状态后，将结果响应发送给 `client`。发送响应完毕后，再次进入到 `conn_new_cmd` 状态，进行连接重置，准备下一次命令处理循环。

在读取、解析、处理、响应过程，遇到任何异常就进入 `conn_closing`，关闭连接。

总结下最近 3 个课时的内容。首先讲解了 Memcached 的原理及特性。然后结合 Memcached 的系统架构，学习了 Mc 基于 Libevent 的多线程网络模型，知道了 Mc 的 IO 主线程负责接受连接及调度，工作线程负责读取指令、处理并响应。本课时还有一个重点是 Memcached 状态机，知道了主线程处理 `conn_listening`，工作线程处理其他 8 种重要状态。每种状态下对应不同的处理逻辑，从而将 Mc 整个冗长复杂的处理过程进行分阶段的处理，每个阶段只关注有限的逻辑，从而确保整个处理过程的清晰、简洁。

最后通过梳理 Mc 命令处理的全过程，学习了 Mc 如何建立连接，如何进行命令读取、处理及响应，从而把 Mc 的系统架构、多线程网络模型、状态机处理进行逻辑打通。

为了方便理解，提供本课时所有知识点的思维导图，如下图所示。

OK，这节课就讲到这里，下一课时我会分享“Memcached 哈希表”，记得按时来听课哈。好，下节课见，拜拜！

## 第四章：Memcached进阶

### 第10讲：MC是怎么定位key的？

你好，我是你的缓存课老师陈波，欢迎你进入第 10 课时“Memcached 哈希表”的学习。

我们在进行 Mc 架构剖析时，除了学习 Mc 的系统架构、网络模型、状态机外，还对 Mc 的 slab 分配、Hashtable、LRU 有了简单的了解。本节课，将进一步深入学习这些知识点。

接下来，进入 Memcached 进阶的学习。会讲解 Mc 是如何进行 key 定位，如何淘汰回收过期失效 key 的，还将分析 Mc 的内存管理 slab 机制，以及 Mc 进行数据存储维护的关键机理，最后还会对 Mc 进行完整的协议分析，并以 Java 语言为例，介绍 Mc 常用的 client，以及如何调优及改进。

key 定位

哈希表

Mc 将数据存储于 Item 中，然后这些 Item 会被 slabclass 的 4 个 LRU 管理。这些 LRU 都是通过双向链表实现数据记录的。双向链表在进行增加、删除、修改位置时都非常高效，但其获取定位 key 的性能非常低下，只能通过链表遍历来实现。因此，Mc 还通过 Hashtable，也就是哈希表，来记录管理这些 Item，通过对 key 进行哈希计算，从而快速定位和读取这些 key/value 所在的 Item，如下图所示。

哈希表也称散列表，可以通过把 key 映射到哈希表中的一个位置来快速访问记录，定位 key 的时间复杂度只有  $O(1)$ 。Mc 的哈希表实际是一个一维指针数组，数组的每个位置称作一个 bucket，即一个桶。性能考虑的需要，Mc 的哈希表的长度设置为 2 的 N 次方。Mc 启动时，默认会构建一个拥有 6.4 万个桶的哈希表，随着新 key 的不断插入，哈希表中的元素超过阈值后，会对哈希表进行扩容，最大可以构建 2 的 32 次方个桶的哈希表，也就是说 Mc 哈希表经过多次扩容后，最多只能有不超过 43 亿个桶。

哈希表设计

对于哈希表设计，有 2 个关键点，一个是哈希算法，一个是哈希冲突解决方案。Mc 使用的哈希算法有 2 种，分别是 Murmur3 Hash 和 Jenkins Hash。Mc 当前版本，默认使用 Murmur3 Hash 算法。不同的 key 通过 Hash 计算，被定位到了相同的桶，这就是哈希冲突。Mc 是通过在每个桶启用一个单向链表，来解决哈希冲突问题的。

定位 key

Memcached 定位 key 时，首先根据 key 采用 Murmur3 或者 Jenkins 算法进行哈希计算，得到一个 32 位的无符号整型输出，存储到变量 hv 中。因为哈希表一般没有  $2^{32}$  那么大，所以需要将 key 的哈希值映射到哈希表的范围内。Mc 采用最简单的取模算法作为映射函数，即采用  $hv \% \text{hashsize}$  进行计算。由于普通的取模运算比较耗时，所以 Mc 将哈希表的长度设置为 2 的 n 次方，采用位运算进行优化，即采用  $hv \& \text{hashmask}$  来计算。hashmask 即 2 的 n 次方减 1。

定位到 key 所在的桶的位置后，如果是插入一个新数据，则将数据 Item 采用头部插入法插入桶的单向链表中。如果是查找，则轮询对应哈希桶中的那个单向链表，依次比对 key 字符串，key 相同则找到数据 Item。

如果哈希表桶中元素太多，这个链表轮询耗时会比较长，所以在哈希表中元素达到桶数的 1.5 倍之后，Mc 会对哈希表进行 2 倍扩容。由于哈希表最多只有 43 亿左右个桶，所以性能考虑，单个 Mc 节点最多存储 65 亿个 key/value。如果要存更多 key，则需要修改 Mc 源码，将最大哈希，即 `HASHPOWER_MAX`，进行调大设置。

哈希表扩容

当 Mc 的哈希表中，Item 数量大于 1.5 倍的哈希桶数量后，Mc 就对哈希表进行扩容处理。如下图所示，Mc 的哈希扩容是通过哈希维护线程进行处理的。准备开始扩容时，哈希维护线程会首先将所有 IO 工作线程和辅助线程进行暂停，其中辅助线程包括 LRU 维护线程、slab 维护线程、LRU 爬虫线程。待这些线程暂停后，哈希维护线程会将当前的主哈希表设为旧哈希表，然后将新的主哈希表扩容之前的 2 倍容量。然后，工作线程及辅助线程继续工作，同时哈希维护线程开始逐步将 Item 元素从旧哈希表迁移到主哈希表。



Mc 在启动时，会根据设置的工作线程数，来构建一个 Item 锁哈希表，线程越多，构建的锁哈希表越大，对于 4 个线程，锁哈希表有 4096 个桶，对于 10 个线程，锁哈希表会有 8192 个桶，Item 锁哈希表最多有 32k 个桶，1k 是 1024，即最多即 32768 个桶。Mc 的锁哈希表中，每个桶对应一个 Item 锁，所以 Mc 最多只有 32768 个 Item 锁。

Mc 哈希表在读取、变更以及扩容迁移过程中，先将 key hash 定位到 Item 锁哈希表的锁桶，然后对 Item 锁进行加锁，然后再进行实际操作。实际上，除了在哈希表，在其他任何时候，只要涉及到在对 Item 的操作，都会根据 Item 中的 key，进行 Item 哈希锁桶加锁，以避免 Item 被同时读写而产生脏数据。Mc 默认有 4096 个锁桶，所以对 key 加锁时，冲突的概率较小，而且 Mc 全部是内存操作，操作速度很快，即便申请时锁被占用，也会很快被释放。

Mc 哈希表在扩容时，哈希表维护线程，每次按桶链表纬度迁移，即一次迁移一个桶里单向链表的所有 Item 元素。在扩容过程中，如果要查找或插入 key，会参照迁移位置选择哈希表。如果 key 对应的哈希桶在迁移位置之前，则到新的主哈希表进行查询或插入，否则到旧哈希表进行查询和插入。待全部扩容迁移完毕，所有的处理就会全部在新的主哈希表进行。

## 第11讲：MC如何淘汰冷key和失效key？

你好，我是你的缓存课老师陈波，欢迎进入第 11 课时“Memcached 淘汰策略”的学习。  
淘汰策略

Mc 作为缓存组件，意味着 Mc 中只能存储访问最频繁的热数据，一旦存入数据超过内存限制，就需要对 Mc 中的冷 key 进行淘汰工作。Mc 中的 key 基本都会有过期时间，在 key 过期后，出于性能考虑，Mc 并不会立即删除过期的 key，而是由维护线程逐步清理，同时，只有这个失效的 key 被访问时，才会进行删除，从而回收存储空间。所以 Mc 对 key 生命周期的管理，即 Mc 对 key 的淘汰，包括失效和删除回收两个纬度，知识结构如下图所示。

key 的失效，包括 key 在 expire 时间之后的过期，以及用户在 flush\_all 之后对所有 key 的过期 2 种方式。

而 Mc 对 key/value 的删除回收，则有 3 种方式。

第一种是获取时的惰性删除，即 key 在失效后，不立即删除淘汰，而在获取时，检测 key 的状态，如果失效，才进行真正的删除并回收存储空间。

第二种方式是在需要对 Item 进行内存分配申请时，如果内存已全部用完，且该 Item 对应的 slabclass 没有空闲的 chunk 可用，申请失败，则会对 LRU 队尾进行同步扫描，回收过期失效的 key，如果没有失效的 key，则会强制删除一个 key。

第三种方式是 LRU 维护线程，不定期扫描 4 个 LRU 队列，对过期 key/value 进行异步淘汰。



## flush\_all

Mc 中，key 失效除了常规的到达过期时间之外，还有一种用 flush\_all 的方式进行全部过期。如果缓存数据写入异常，出现大量脏数据，而又没有简单的办法快速找出所有的脏数据，可以用 flush\_all 立即让所有数据失效，通过 key 重新从 DB 加载的方式来保证数据的正确性。flush\_all 可以让 Mc 节点的所有 key 立即失效，不过，在某些场景下，需要让多个 Mc 节点的数据在某个时间同时失效，这时就可以用 flush\_all 的延迟失效指令了。该指令通过 flush\_all 指令后面加一个 expiretime 参数，可以让多个 Mc 在某个时间同时失效所有的 key。

flush\_all 后面没有任何参数，等价于 flush\_all 0，即立即失效所有的 key。当 Mc 收到 flush\_all 指令后，如果是延迟失效，会将全局 setting 中的 oldest\_live 设为指定 N 秒后的时间戳，即 N 秒后失效；如果是立即失效，则将全局 setting 中的 oldest\_cas 设为当前最大的全局 cas 值。设置完这个全局变量值后，立即返回。因此，在 Mc 通过 flush\_all 失效所有 key 时，实际不做任何 key 的删除操作，这些 key，后续会通过用户请求同步删除，或 LRU 维护线程的异步删除，来完成真正的删除动作。

### 惰性删除

Mc 中，过期失效 key 的惰性主动删除，是指在 touch、get、gets 等指令处理时，首先需要查询 key，找到 key 所在的 Item，然后校验 key 是否过期，是否被 flush，如果过期或被 flush，则直接进行真正的删除回收操作。

对于校验 key 过期很容易，直接判断过期时间即可。对于检查 key 是否被 flush，处理逻辑是首先检查 key 的最近访问时间是否小于全局设置中的 oldest\_live，如果小于则说明 key 被 flush 了；否则，再检查 key 的 cas 唯一 id 值，如果小于全局设置中的 oldest\_cas，说明也被 flush 了。

### 内存分配失败，LRU 同步淘汰

Mc 在插入或变更 key 时，首先会在适合的 slabclass 为新的 key/value 分配一个空闲的 Item 空间，如果分配失败，会同步对该 slabclass 的 COLD LRU 进行队尾元素淘汰，如果淘汰回收成功，则 slabclass 会多一个空闲的 Item，这个 Item 就可以被前面那个 key 来使用。如果 COLD LRU 队列没有 Item 数据，则淘汰失败，此时会对 HOT LRU 进行队尾轮询，如果 key 过期失效则进行淘汰回收，否则进行迁移。

### LRU 维护线程，异步淘汰

在 key 进行读取、插入或变更时，同步进行 key 淘汰回收，并不是一种高效的办法，因为淘汰回收操作相比请求处理，也是一个重量级操作，会导致 Mc 性能大幅下降。因此 Mc 额外增加了一个 LRU 维护线程，对过期失效 key 进行回收，在不增加请求负担的情况下，尽快回收失效 key 锁占用的空间。

前面讲到，Mc 有 64 个 slabclass，其中 1~63 号 slabclass 用于存取 Item 数据。实际上，为了管理过期失效数据，1~63 号 slabclass 还分别对应了 4 个 LRU，分布是 TEMP、HOT、WARM、COLD LRU。所以这就总共有  $63 \times 4 = 252$  个 LRU。LRU 维护线程，会按策略间断 sleep，待 sleep 结束，就开始对 4 个 LRU 进行队尾清理工作。

Mc 在新写入 key 时，如果 key 的过期时间小于 61s，就会直接插入到 TEMP LRU 中，如下图所示。TEMP LRU 没有长度限制，可以一直插入，同时因为过期时间短，TEMP LRU 不进行队列内部的搬运和队列间的迁移，确保处理性能最佳。LRU 维护线程在 sleep 完毕后，首先会对 TEMP LRU 队尾进行 500 次轮询，然后在每次轮询时，会进行 5 次小循环。小循环时，首先检查 key 是否过期失效，如果失效则进行回收淘汰，然后继续小循环；如果遇到一个没失效的 key，则回收该 key 并退出 TEMP LRU 的清理工作。如果 TEMP LRU 队尾 key 全部失效，维护线程一次可以回收  $500 \times 5$  共 2500 个失效的 key。

如下图，MC 在新写入 key 时，如果 key 的过期时间超过 61s，就会直接插入到 HOT LRU。HOT LRU 会有内存限制，每个 HOT LRU 所占内存不得超过所在 slabclass 总实际使用内存的 20%。LRU 维护线程在执行日常维护工作时，首先对 TEMP LRU 进行清理，接下来就会对 HOT LRU 进行维护。HOT LRU 的维护，也是首先轮询 500 次，每次轮询进行 5 次小循环，小循环时，首先检查 key 是否过期失效，如果失效则进行回收淘汰，然后继续小循环。直到遇到没失效的 key。如果这个 key 的状态是 ACTIVE，则迁移到 WARM LRU。对于非 ACTIVE 状态的 key，如果 HOT LRU 内存占用超过限制，则迁移到 COLD LRU，否则进行纾困性清理掉该 key，注意这种纾困性清理操作一般不会发生，一旦发生，虽然会清理掉该 key，但操作函数此时也认定本次操作回收和清理 keys 数仍然为 0。

如下图，如果 HOT LRU 中回收和迁移的 keys 数为 0，LRU 维护线程会对 WARM LRU 进行轮询。WARM LRU 也有内存限制，每个 WARM LRU 所占内存不得超过所在 slabclass 总实际使用内存的 40%。WARM LRU 的维护，也是首先轮询 500 次，每次轮询进行 5 次小循环，小循环时，首先检查 key 是否过期失效，如果失效则进行回收淘汰，然后继续小循环。直到遇到没失效的 key。如果这个 key 的状态是 ACTIVE，则内部搬运到 LRU 队列头部。对于非 ACTIVE 状态的 key，如果 WARM LRU 内存占用超过限制，则迁移到 COLD LRU，否则进行纾困性清理掉该 key。注意这种纾困性清理操作一般不会发生，一旦发生，虽然会清理掉该 key，但操作函数此时也认定本次操作回收和清理 keys 数仍然为 0。

LRU 维护线程最后会对 COLD LRU 进行维护，如下图。与 TEMP LRU 相同，COLD LRU 也没有长度限制，可以持续存放数据。COLD LRU 的维护，也是首先轮询 500 次，每次轮询进行 5 次小循环，小循环时，首先检查 key 是否过期失效，如果失效则进行回收淘汰，然后继续小循环。直到遇到没失效的 key。如果这个 key 的状态是 ACTIVE，则会迁移到 WARM LRU 队列头部，否则不处理直接返回。

LRU 维护线程处理时，TEMP LRU 是在独立循环中进行，其他三个 LRU 在另外一个循环中进行，如果 HOT、WARM、COLD LRU 清理或移动的 keys 数为 0，则那个 500 次的大循环就立即停止。

## 第12讲：为何MC能长期维持高性能读写？

你好，我是你的缓存课老师陈波，欢迎进入第 12 课时“Memcached 内存管理 slab 机制”的学习。  
内存管理 slab 机制

讲完淘汰策略，我们接下来学习内存管理 slab 机制。

Mc 内存分配采用 slab 机制，slab 机制可以规避内存碎片，是 Mc 能持续高性能进行数据读写的关键。  
slabclass

Mc 的 slab 机制是通过 slabclass 来进行运作的，如下图所示。Mc 在启动时，会构建长度为 64 的 slabclass 数组，其中 0 号 slabclass 用于 slab 的重新分配，1~63 号 slabclass 存储数据 Item。存储数据的每个 slabclass，都会记录本 slabclass 的 chunk size，同时不同 slabclass 的 chunk size 会按递增因子增加，最后一个 slabclass（即 63 号 slabclass）的 chunk size 会直接设为最大的 chunk size，默认是 0.5MB。每个 slabclass 在没有空闲的 chunk 时，Mc 就会为其分配一个默认大小为 1MB 的

slab，同时按照本 slabclass 的 chunk size 进行拆分，这些分拆出来的 chunk 会按 Item 结构体进行初始化，然后记录到 slabclass 的 freelist 链表中。当有 key/value 要存储在本 slabclass 时，就从 freelist 分配一个 Item，供其使用。同时，如果 Item 过期了，或被 flush\_all 失效了，或在内存不够时被强项剔除了，也会在适当时刻，重新被回收到 freelist，以供后续分配使用。

## 存储 slab 分配

如下图所示，Mc 的存储空间分配是以 slab 为单位的，每个 slab 的默认大小时 1MB。因此在存数据时，Mc 的内存最小分配单位是 1MB，分配了这个 1MB 的 slab 后，才会进一步按所在 slabclass 的 chunk size 进行细分，分拆出的相同 size 的 chunk。这个 chunk 用来存放 Item 数据，Item 数据包括 Item 结构体字段，以及 key/value。

一般来讲，Item 结构体及 key/value 不会填满 chunk，会存在少量字节的浪费，但这个浪费的字节很少，基本可以忽略。Mc 中，slab 一旦分配，就不会再被回收，但会根据运行状况，重新在不同 slabclass 之间进行分配。

当一个 slabclass 没有空闲 chunk，而新数据插入时，就会对其尝试增加一个新的 slab。slabclass 增加新 slab 时，首先会从 0 号全局 slabclass 中复用之前分配的 slab，如果 0 号 slabclass 没有 slab，则会尝试从内存堆空间直接分配一个 slab。如果 0 号全局 slabclass 没有空闲 slab，而且 Mc 内存分配已经达到 Mc 设定的上限值，就说明此时没有可重新分配的 slab，分配新 slab 失败，直接返回。

当然，虽然 slabclass 分配 slab 失败，但并不意味着 Item 分配会失败，前面已经讲到，可以通过同步 LRU 淘汰，回收之前分配出去的 Item，供新的存储请求使用。

Item

Mc 中，slabclass 中的 chunk 会首先用 Item 结构体进行初始化，然后存到 freelist 链表中，待需要分配给数据存储时，再从 freelist 中取出，存入 key/value，以及各种辅助属性，然后再存到 LRU 链表及 Hashtable 中，如下图所示。Item 结构体，首先有两个 prev、next 指针，在分配给待存储数据之前，这两个指针用来串联 freelist 链表，在分配之后，则用来串联所在的 LRU 链表。接下来是一个 h\_next 指针，用来在分配之后串联哈希表的桶单向链表。Item 结构体还存储了过期时间、所属 slabclass id，key 长度、cas 唯一 id 值等，最后在 Item 结构体尾部，存储了 key、flag、value 长度，以及 value block 数据。在 value 之后的 chunk 空间，就被浪费掉了。Item 在空闲期间，即初始分配时以及被回收后，都被 freelist 管理。在存储期间，被哈希表、LRU 管理。

## 存储 Item 分配

Mc 采用 slab 机制管理分配内存，采用 Item 结构存储 key/value，因此对存储 key/value 的内存分配，就转换为对 Item 的分配。分配 Item 空间时，会进行 10 次大循环，直到分配到 Item 空间才会提前返回。如果循环了 10 次，还没有分配到 Item 空间，则存储失败，返回一个 SERVER\_ERROR 响应。

在分配过程中，首先，如果 slabclass 的 freelist 有空间，则直接分配。否则，尝试分配一个新的 slab，新 slab 依次尝试从全局 slab 池（即 0 号 slabclass）中复用空闲 slab，如果全局 slab 池没有 slab，则尝试从内存直接分配。分配新 slab 成功后，会按照 slabclass 记录的 chunk size 对 slab 进行分拆，并将分拆出来的 chunk 按 Item 结构初始化后记录到 freelist。如果全局 slab 池为空，且 Mc 内存分配已经达到设定的上限，则走新增 slab 的路径失败，转而进行 5 次小循环，尝试从 COLD LRU 回收过期 key，如果没有过期则直接强制剔除队尾的一个正常 key。如果该 slabclass 的 COLD LRU 没有 Item，则对其 HOT LRU 进行处理，对 HOT 链表队尾 Item 进行回收或者迁移，以方便在下次循环中找到一个可用的 Item 空间。

## 数据存储机理

讲完 Mc 的哈希表定位、LRU 淘汰、slab 内存分配，接下来我们来看看 Mc 中 key/value 数据的存储机理，通过对数据存储以及维护过程的分析，来把 Mc 的核心模块进行打通和关联。

首先来看 Mc 如何通过 slab 机制将数据写入预分配的存储空间。

如下图所示，当需要存储 key/value 数据时，首先根据 key/value size，以及 Item 结构体的 size，计算出存储这个 key/value 需要的字节数，然后根据这个字节数选择一个能存储的 chunk size 最小的 slabclass。再从这个 slabclass 的 freelist 分配一个空闲的 chunk 给这个 key/value 使用。如果 freelist 为空，首先尝试为该 slabclass 新分配一个 slab，如果 slab 分配成功，则将 slab 按 size 分拆出一些 chunk，通过 Item 结构初始化后填充到 freelist。如果 slab 分配失败，则通过 LRU 淘汰失效的 Item 或强行剔除一个正常的 Item，然后这些 Item 也会填充到 freelist。当 freelist 有 Item 时，即可分配给 key/value。这个过程会重试 10 次，直到分配到 Item 位置。一般情况下，Item 分配总会成功，极小概率情况下也会分配失败，如果分配失败，则会回复一个 SERVER\_ERROR 响应，通知 client 存储失败。分配到一个空闲的 Item 后，就会往这个 Item 空间写入过期时间、flag、slabclass id、key，以及 value 等。对于 set 指令，如果这个 key 还有一个旧值，在存入新 value 之前，还会先将这个旧值删除掉。

当对 key/value 分配 Item 成功，并写入数据后，接下来就会将这个 Item 存入哈希表。因为 Mc 哈希表存在迁移的情况，所以对于正常场景，直接存入主哈希表。在哈希表迁移期间，需要根据迁移位置，选择存入主哈希表还是旧哈希表。存入哈希表之后，这个 key 就可以快速定位了。然后这个 Item 还会被存入 LRU，Mc 会根据这个 key 的过期时间进行判断，如果过期时间小于 61s，则存入 TEMP LRU，否则存入 HOT LRU。

至此，这个 key/value 就被正确地存入 Mc 了，数据内容写入 slabclass 中某个 slab 的 chunk 位置，该 chunk 用 Item 结构填充，这个 Item 会被同时记录到 Hashtable 和 LRU，如下图所示。通过 Hashtable 可以快速定位查到这个 key，而 LRU 则用于 Item 生命周期的日常维护。

Mc 对 Item 生命周期的日常维护，包括异步维护和同步维护。异步维护是通过 LRU 维护线程来进行的，整个过程不影响 client 的正常请求，在 LRU 维护线程内，对过期、失效 key 进行回收，并对 4 个 LRU 进行链表内搬运和链表间迁移。这是 Item 生命周期管理的主要形式。同步维护，由工作线程在处理请求命令时进行。工作线程在处理 delete 指令时，会直接将 key/value 进行删除。在存储新 key/value 时，如果分配失败，会进行失效的 key 回收，或者强行剔除正常的 Item。这些 Item 被回收后，会进入到 slabclass 的 freelist 进行重复使用。

## 第13讲：如何完整学习MC协议及优化client访问？

你好，我是你的缓存课老师陈波，欢迎进入第 13 课时“Memcached 协议分析”的学习。

协议分析

异常错误响应

接下来，我们来完整学习 Mc 协议。在学习 Mc 协议之前，首先来看看 Mc 处理协议指令，如果发现异常，如何进行异常错误响应的。Mc 在处理所有 client 端指令时，如果遇到错误，就会返回 3 种错误信息中的一种。

第一种错误是协议错误，一个"ERROR\r\n"的字符串。表明 client 发送了一个非法命令。

第二种错误是 client 错误，格式为"CLIENT\_ERROR <error-描述信息>\r\n"。这个错误信息表明，client 发送的协议命令格式有误，比如少了字段、多了非法字段等。

第三种错误是"SERVER\_ERROR <error-描述信息>\r\n"。这个错误信息表明 Mc server 端，在处理命令时出现的错误。比如在给 key/value 分配 Item 空间失败后，会返回"SERVER\_ERROR out of memory storing object" 错误信息。

## 存储协议命令

现在再来看看 Mc 的存储协议。Mc 的存储协议命令不多，只有 6 个。

Mc 存储指令分 2 行。第一行是报文首部，第二行是 value 的 data block 块。这两部分用 \r\n 来进行分割和收尾。

存储类指令的报文首行分 2 种格式，其中一种是在 cmd 存储指令，后面跟 key、flags、expiretime、value 字节数，以及一个可选的 noreply。

其中 flags 是用户自己设计的一个特殊含义数字，Mc 对 flag 只存储，而不进行任何额外解析处理，expiretime 是 key 的过期时间，value 字节数是 value block 块的字节长度，而带上 noreply 是指 Mc 处理完后静默处理，不返回任何响应给 client。

这种 cmd 指令包括我们最常用的 set 指令，另外还包括 add、replace、append、reppend，总共 5 个指令：

Set 命令用于存储一个 key/value；

Add 命令是在当 key 不存在时，才存储这个 key/value；

Replace 命令，是当 key 存在时，才存储这个 key/value；

Append 命令，是当 key 存在时，追加 data 到 value 的尾部；

Prepend 命令，是当 key 存在时，将 data 加到 value 的头部。

另外一种存储协议指令，主要格式和字段与前一种基本相同，只是多了一个 cas unique id，这种格式只有 cas 指令使用。cas 指令是指只有当这个 key 存在，且从本 client 获取以来，没有其他任何人修改过时，才进行修改。cas 的英文含义是 compare and set，即比较成功后设置的意思。

## 存储命令响应

Mc 在响应存储协议时，如果遇到错误，就返回前面说的3种错误信息中的一种。否则就会返回如下 4 种正常的响应，"STORED\r\n"、"EXISTS\r\n"、"NOT\_STORED\r\n"、"NOT\_FOUND\r\n"。

其中，stored 表明存储修改成功。NOT\_STORED 表明数据没有存储成功，但并不是遇到错误或异常。这个响应一般表明 add 或 replace 等指令，前置条件不满足时，比如 add，这个 key 已经存在 Mc，就会 add 新 key 失败。replace 时，key 不存在，也无法 replace 成功。EXISTS 表明待 cas 的 key 已经被修改过了，而 NOT\_FOUND 是指待 cas 的 key 在 Mc 中不存在。

Mc 对存储命令的请求及响应协议，可以参考下面的思维导图来有一个完整的印象。

### 获取命令

Mc 的获取协议，只有 get、gets 两种指令，如下图所示。格式为 get/gets 后，跟随若干个 key，然后 \r\n 结束请求命令。get 指令只获取 key 的 flag 及 value，gets 会额外多获取一个 cas unique id 值。gets 主要是为 cas 指令服务的。

获取命令的响应，就是 value 字符串，后面跟上 key、flag、value 字节数，以及 value 的 data block 块。最后跟一个 END\r\n 表明所有存在的 key/value 已经返回，如果没有返回的 key，则表明这个 key 在 Mc 中不存在。

### 其他指令

Mc 的其他协议指令包括 delete、incr、decr、touch、gat、gats、slabs、lru、stats 这 9 种指令。

其中 delete 用于删除一个 key。

incr/decr 用于对一个无符号长整型数字进行加或减。

touch、gat、gats 是 Mc 后来增加的指令，都可以用来修改 key 的过期时间。不同点是 touch 只修改 key 的过期时间，不获取 key 对应的 value。

而 gat、gats 指令，不仅会修改 key 的过期时间，还会获取 key 对应的 flag 和 value 数据。gats 同 gets，还会额外获取 cas 唯一 id 值。

Slabs reassign 用于在 Mc 内存达到设定上限后，将 slab 重新在不同的 slabclass 之间分配。这样可以规避 Mc 启动后自动分配而产生随机性，使特殊 size 的数据也得到较好的命中率。Slabs automove 是一个开关指令，当打开时，就允许 Mc 后台线程自行决定何时将 slab 在 slabclass 之间重新分配。

Lru 指令用于 Mc LRU 的设置和调优。比如 LRU tune 用于设置 HOT、WARM LRU 的内存占比。LRU mode 用来设置 Mc 只使用 COLD LRU，还是使用新版 4 个 LRU 的新策略。LRU TEMP\_TTL 用来设置 Mc 的 TEMP LRU 的 TTL 值，默认是 61s，小于这个 TEMP\_TTL 的 key 会被插入到 TEMP LRU。

Stats 用于获取 Mc 的各种统计数据。Stats 后面可以跟 statistics、slabs、size 等参数，来进一步获取更多不同的详细统计。

### Client 使用

Mc 在互联网企业应用广泛，热门语言基本都有 Mc client 的实现。以 Java 语言为例，互联网业界广泛使用的有 Memcached-Java-Client、SpyMemcached、Xmemcached 等。

Memcached-Java-Client 推出时间早，10 年前就被广泛使用，这个 client 性能一般，但足够稳定，很多互联网企业至今仍在使用。不过这个 client 几年前就停止了更新。

SpyMemcached 出现的比较晚，性能较好，但高并发访问场景，稳定性欠缺。近几年变更很少，基本停止了更新。

Xmemcached 性能较好，综合表现最佳。而且社区活跃度高，近些年也一直在持续更新中。Java 新项目启动，推荐使用 Xmemcached。

在使用 Mc client 时，有一些通用性的调优及改进方案。比如，如果读写的 key/value 较大，需要设置更大的缓冲 buf，以提高性能。在一些业务场景中，需要启用 TCP\_NODELAY，避免 40ms 的延迟问题。同时，如果存取的 key/value size 较大，可以设置一个压缩阈值，超过阈值，就对 value 进行压缩算法，减少读写及存储的空间。

为了避免缓存雪崩，并更好地应对极热 key 及洪水流量的问题，还可以对 Mc client 进行封装，加入多副本、多层级策略，使 Mc 缓存系统在任何场景下，都可做到高可用、高性能。

讲到这里，Mc 的核心知识点就基本讲完了，知识点结构图如下所示。

回顾一下最近几节课的内容。首先，学习了 Mc 的系统架构，学习了 Mc 基于 libevent 的网络模型，学习了 Mc 的多线程处理，包括主线程、工作线程如何进行网络 IO 协调及处理，学习了 Mc 的状态机。然后，继续学习了 Mc 用于定位 key 的哈希表，学习了用于数据生命周期管理的 LRU，还学习 slab 分配机制，以及 Mc 数据的存储机理。最后，还完整学习了 Mc 的协议，了解了以 Java 语言为例的 3 种 Mc client，以及 Mc client 在线上使用过程中，如何进行调优及改进。

根据下面 Mc 协议的思维导图，查看自己是否对所有指令都有理解，可以结合 Mc 的协议文档，启动一个 Mc 实例，进行各个命令的实际操练。

OK，这节课就讲到这里啦，下一课时我将分享“Memcached 经典问题及解决方案”，记得按时来听课哈。好，下节课见，拜拜！

## 第五章：分布式Memcached实战

---

### 第14讲：大数据时代，MC如何应对新的常见问题？

你好，我是你的缓存课老师陈波，欢迎进入第 14 课时“Memcached 经典问题及解决方案”的学习。  
大数据时代 Memcached 经典问题

随着互联网的快速发展和普及，人类进入了大数据时代。在大数据时代，移动设备全面融入了人们的工作和生活，各种数据以前所未有的速度被生产、挖掘和消费。移动互联网系统也不断演进和发展，存储、计算和分析这些海量数据，以满足用户的需要。在大数据时代，大中型互联网系统具有如下特点。

首先，系统存储的数据量巨大，比如微博系统，每日有数亿条记录，历史数据达百亿甚至千亿条记录。

其次，用户多，访问量巨大，每日峰值流量高达百万级 QPS。

要存储百千亿级的海量数据，同时满足大量用户的高并发访问，互联网系统需要部署较多的服务实例，不少大中型互联网系统需要部署万级，甚至十万级的服务实例。

再次，由于大数据时代，社会信息获取扁平化，热点事件、突发事件很容易瞬间引爆，引来大量场外用户集中关注，从而形成流量洪峰。

最后，任何硬件资源都有发生故障的概率，而且存在 4 年故障效应，即服务资源在使用 4 年后，出现故障的概率会陡增；由于大中型互联网系统的部署，需要使用大量的服务器、路由器和交换机，同时部署在多个地区的不同 IDC，很多服务资源的使用时间远超 4 年，局部出现硬件故障、网络访问异常就比较常见了。

由于互联网系统会大量使用 Memcached 作为缓存，而在使用 Memcached 的过程中，同样也会受到前面所说的系统特点的影响，从而产生特有的经典问题。

#### 容量问题

第一个问题是容量问题。Memcached 在使用中，除了存储数据占用内存外，连接的读写缓冲、哈希表分配、辅助线程处理、进程运行等都会占用内存空间，而且操作系统本身也会占用不少内存，为了确保 Mc 的稳定运行，Mc 的内存设置，一般设为物理内存的 80%。另外，设置的内存，也不完全是存储有效数据，我上一节课讲到，每个 Item 数据存储在 chunk 时，会有部分字节浪费，另外 key 在过期、失效后，不是立即删除，而是采用延迟淘汰、异步 LRU 队尾扫描的方式清理，这些暂时没有淘汰的、过期失效的 key，也会占用不少的存储空间。当前大数据时代，互联网系统中的很多核心业务，需要缓存的热数据在 300~500GB 以上，远远超过单机物理内存的容量。

#### 性能瓶颈

第二个问题是性能瓶颈问题。出于系统稳定性考虑，线上 Mc 的访问，最大 QPS 要在 10~20w 以下，超过则可能会出现慢查的问题。而对中大型互联网系统，核心业务的缓存请求高达百万级 QPS，仅仅靠简单部署单个物理机、单个资源池很难达到线上的业务要求。

#### 连接瓶颈

第三个问题是连接瓶颈的问题。出于稳定性考虑，线上 Mc 的连接数要控制在 10w 以下。以避免连接数过多，导致连接占用大量内存，从而出现命中率下降、甚至慢查超时的问题。对于大中型系统，线上实例高达万级、甚至十万级，单个实例的最小、最大连接数，一般设置在 5~60 个之间。业务实例的连接数远超过单个机器的稳定支撑范围。

#### 硬件资源局部故障

第四个问题是硬件资源局部故障，导致的缓存体系的可用性问题。由于任何硬件资源，都有一定故障概率，而且在使用 4 年后，故障率陡增。对于数以万计的硬件设备，随时都有可能出现机器故障，从而导致 Mc 节点访问性能下降、宕机，海量访问穿透到 DB，引发 DB 过载，最终导致整个系统无法访问，引发雪崩现象。

#### 流量洪峰下快速扩展



第五个问题是在流量洪峰的场景下，如何快速扩展的问题。大数据时代，由于信息扩散的扁平化，突发事件、重大活动发生时，海量用户同时蜂拥而至，短时间引发巨大流量。整个系统的访问量相比日常峰值增大 70% 以上，同时出现大量的极热 key 的访问，这些极热 key 所在的 Mc 节点，访问量相比日常高峰，增大 2~3 倍以上，很容易出现 CPU 飙升、带宽打满、机器负荷严重过载的现象。

#### Memcached 经典问题及应对方案

为了解决大中型互联网系统在使用 Mc 时的这些问题。我们可以使用下面的解决方案。

#### Memcached 分拆缓存池

首先对系统内的核心业务数据进行分拆，让访问量大的数据，使用独立的缓存池。同时每个缓存池 4~8 个节点，这样就可以支撑足够大的容量，还避免单个缓存节点压力过大。对于缓存池的分布策略，可以采用一致性哈希分布和哈希取模分布。

一致性哈希分布算法中，首先计算 Mc 服务节点的哈希值，然后将其持续分散配置在圆中，这样每个缓存节点，实际包括大量大小各异的 N 个 hash 点。如下图所示，在数据存储或请求时，对 key 采用相同的 hash 算法，并映射到前面的那个圆中，从映射位置顺时针查找，找到的第一个 Mc 节点，就是目标存取节点。

而哈希取模分布算法，则比较简单，对 key 做 hash 后，对 Mc 节点数取模，即可找到待存取的目标 Mc 节点。

系统运行过程中，Mc 节点故障不可避免，有时候甚至短期内出现多次故障。在 Mc 节点故障下线后，如果采用一致性 hash 分布，可以方便得通过 rehash 策略，将该 Mc 节点的 hash 点、访问量，均匀分散到其他 Mc 节点。如果采用取模分布，则会直接导致 1/N 的访问 miss，N 是 Mc 资源池的节点数。

因此，对于单层 Mc 缓存架构，一致性 hash 分布配合 rehash 策略，是一个更佳方案。通过将业务数据分拆到独立 Mc 资源池，同时每个资源池采用合适的分布算法，可以很好的解决 Mc 使用中容量问题、性能瓶颈问题，以及连接瓶颈问题。

#### Master-Slave 两级架构

在系统的访问量比较大，比如峰值 QPS 达到 20w 以上时，如果缓存节点故障，即便采用一致性 hash，也会在一段时间内给 DB 造成足够大的压力，导致大量慢查询和访问超时的问题。另外，如果某些缓存服务器短期多次故障，反复上下线，多次 rehash 还会产生脏数据。对此，可以采用 Master-Slave 的两级架构方案。

在这种架构方案下，将业务正常访问的 Memcached 缓存池作为 master，然后在 master 之后，再加一个 slave 资源池作 master 的热备份。slave 资源池也用 6~8 个节点，内存设置只用 master 的 1/2~1/3 即可。因为 slave 的应用，主要是考虑在 master 访问 miss 或异常时，Mc 缓存池整体的命中率不会过度下降，所以并不需要设置太大内存。

日常访问，对于读操作，直接访问 master，如果访问 miss，再访问 slave。如果 slave 命中，就将读取到的 key 回写到 master。对于写操作，set、touch 等覆盖类指令，直接更新 master 和 slave；而 cas、append 等，以 master 为准，master 在 cas、add 成功后，再将 key 直接 set 到 slave，以保持 master、slave 的数据一致性。

如下图，在 master 部分节点异常后，由 slave 层来承接。任何一层，部分节点的异常，不会影响整体缓存的命中率、请求耗时等 SLA 指标。同时分布方式采用哈希取模方案，mc 节点异常不 rehash，直接穿透，方案简洁，还可以避免一致性 hash 在 rehash 后产生的脏数据问题。

Master-Slave 架构，在访问量比较大的场景下，可以很好得解决局部设备故障的问题。在部分节点异常或访问 miss 时，多消耗 1ms 左右的时间，访问 slave 资源，实现以时间换系统整体可用性的目的。

#### M-S-L1 架构

20世纪初，意大利统计学家帕累托提出一个观点：在任何特定群体中，重要的因子通常只占少数，而不重要的因子则占多数，因此只要能控制具有重要性的少数因子，即能控制全局。这个理论经过多年演化，就成为当前大家所熟悉的 80/20 定律。80/20 定律在互联网系统中也广泛存在，如 80% 的用户访问会集中在系统 20% 的功能上，80% 的请求会集中在 20% 的数据上。因此，互联网系统的数据，有明显的冷热区分，而且这个冷热程度往往比 80/20 更大，比如微博、微信最近一天的数据，被访问的特别频繁，而一周前的数据就很少被访问了。而且最近几天的热数据中，部分 feed 信息会被大量传播和交互，比其他大部分数据的访问量要高很多倍，形成明显的头部请求。

头部请求，会导致日常大量访问，被集中在其中一小部分 key 上。同时，在突发新闻、重大事件发生时，请求量短期增加 50~70% 以上，而这些请求，又集中在 突发事件的关联 key 上，造就大量的热 key 的出现。热 key 具有随机性，如果集中在某少数几个节点，就会导致这些节点的压力陡增数倍，负荷严重过载，进而引发大量查询变慢超时的问题。

为了应对日常峰值的热数据访问，特别是在应对突发事件时，洪峰流量带来的极热数据访问，我们可以通过增加 L1 层来解决。如下图所示，L1 层包含 2~6 组 L1 资源池，每个 L1 资源池，用 4~6 个节点，但内存容量只要 Master 的 1/10 左右即可。

如图，读请求时，首先随机选择一个 L1 进行读取，如果 miss 则访问 master，如果 master 也 miss，最后访问 slave。中途，只要任何一层命中，则对上一层资源池进行回写。

写请求时，同 Master-Slave 架构类似，对于 set 覆盖类指令，直接 set 三层所有的资源池。对于 add/cas/append 等操作，以 master 为准，master 操作成功后，将最后的 key/value set 到 L1 和 slave 层所有资源池。

由于 L1 的内存只有 master 的 1/10，且 L1 优先被读取，所以 L1 中 Memcached 只会保留最热的 key，因为 key 一旦稍微变冷，就会排到 COLD LRU 队尾，并最终被剔除。虽然 L1 的内存小，但由于 L1 里，永远只保存了 系统访问量 最大最热的数据，根据我们的统计，L1 可以满足整个系统的 60~80% 以上的请求数据。这也与 80/20 原则相符合。

master 存放全量的热数据，用于满足 L1 读取 miss 或异常后的访问流量。slave 用来存放绝大部分的热数据，而且与 master 存在一定的差异，用来满足 L1、master 读取 miss 或异常的访问流量。

这里面有个可以进一步优化的地方，即为确保 master、slave 的热度，让 master、slave 也尽可能只保留最热的那部分数据，可以在读取 L1 时，保留适当的概率，直接读取 master 或 slave，让最热的 key 被访问到，从而不会被 master、slave 剔除。此时，访问路径需要稍做调整，即如果首先访问了 master，如果 miss，接下来只访问 slave。而如果首先访问了 slave，如果 miss，接下来只访问 master。

通过 Master-Slave-L1 架构，在流量洪峰到来之际，我们可以用很少的资源，快速部署多组 L1 资源池，然后加入 L1 层中，从而让整个系统的抗峰能力达到 N 倍的提升。从而以最简洁的办法，快速应对流量洪峰，把极热 key 分散到 N 组 L1 中，每个 L1 资源池只用负责 1/N 的请求。除了抗峰，另外，还可以轻松应对局部故障，避免雪崩的发生。

本课时，讲解了大数据时代下大中型互联网系统的特点，访问 Memcached 缓存时的经典问题及应对方案；还讲解了如何通过分拆缓存池、Master-Slave 双层架构，来解决 Memcached 的容量问题、性能瓶颈、连接瓶颈、局部故障的问题，以及 Master-Slave-L1 三层架构，通过多层、多副本 Memcached 体系，来更好得解决突发洪峰流量和局部故障的问题。

可以参考下面的思维导图，对这些知识点进行回顾和梳理。

OK，这节课就讲到这里啦，下一课时我将分享“Twemproxy 框架、应用及扩展”相关的知识，记得按时来听课哈。好，下节课见，拜拜！

## 第15讲：如何深入理解、应用及扩展 Twemproxy？

你好，我是你的缓存课老师陈波，欢迎进入第 15 课时“Twemproxy 框架、应用及扩展”的学习。

Twemproxy 架构及应用

Twemproxy 是 Twitter 的一个开源架构，它是一个分片资源访问的代理组件。如下图所示，它可以封装资源池的分布及 hash 规则，解决后端部分节点异常后的探测和重连问题，让 client 访问尽可能简单，同时资源变更时，只要在 Twemproxy 变更即可，不用更新数以万计的 client，让资源变更更轻量。最后，Twemproxy 跟后端通过单个长连接访问，可以大大减少后端资源的连接压力。

系统架构

接下来分析基于 Twemproxy 的应用系统架构，以及 Twemproxy 组件的内部架构。

如下图所示，在应用系统中，Twemproxy 是一个介于 client 端和资源端的中间层。它的后端，支持 Memcached 资源池和 Redis 资源池的分片访问。Twemproxy 支持取模分布和一致性 hash 分布，还支持随机分布，不过使用场景较少。

应用前端在请求缓存数据时，直接访问 Twemproxy 的对应端口，然后 Twemproxy 解析命令得到 key，通过 hash 计算后，按照分布策略，将 key 路由到后端资源的分片。在后端资源响应后，再将响应结果返回给对应的 client。

在系统运行中，Twemproxy 会自动维护后端资源服务的状态。如果后端资源服务异常，会自动进行剔除，并定期探测，在后端资源恢复后，再对缓存节点恢复正常使用。

#### 组件架构

Twemproxy 是基于 epoll 事件驱动模型开发的，架构如下图所示。它是一个单进程、单线程组件。核心进程处理所有的事件，包括网络 IO，协议解析，消息路由等。Twemproxy 可以监听多个端口，每个端口接受并处理一个业务的缓存请求。Twemproxy 支持 Redis、Memcached 协议，支持一致性 hash 分布、取模分布、随机分布三种分布方案。Twemproxy 通过 YAML 文件进行配置，简单清晰，且便于人肉读写。

Twemproxy 与后端资源通过单个长连接访问，在收到业务大量并发请求后，会通过 pipeline 的方式，将多个请求批量发到后端。在后端资源持续访问异常时，Twemproxy 会将其从正常列表中剔除，并不断探测，待其恢复后再进行请求的路由分发。

Twemproxy 运行中，会持续产生海量请求及响应的消息流，于是开发者精心设计了内存管理机制，尽可能的减少内存分配和复制，最大限度的提升系统性能。Twemproxy 内部，请求和响应都是一个消息，而这个消息结构体，以及消息存放数据的缓冲都是重复使用的，避免反复分配和回收的开销，提升消息处理的性能。为了解决短连接的问题，Twemproxy 的连接也是复用的，这样在面对 PHP client 等短连接访问时，也可以反复使用之前分配的 connection，提升连接性能。

另外，Twemproxy 对消息还采用了 zero copy（即零拷贝）方案。对于请求消息，只在 client 接受时读取一次，后续的解析、处理、转发都不进行拷贝，全部共享最初的那个消息缓冲。对于后端的响应也采用类似方案，只在接受后端响应时，读取到消息缓冲，后续的解析、处理及回复 client 都不进行拷贝。通过共享消息体及消息缓冲，虽然 Twemproxy 是单进程/单线程处理，仍然可以达到 6~8w 以上的 QPS。

#### Twemproxy 请求及响应

接下来看一下 Twemproxy 是如何进行请求路由及响应的。

Twemproxy 监听端口，当有 client 连接进来时，则 accept 新连接，并构建初始化一个 client\_conn。当建连完毕，client 发送数据到来时，client\_conn 收到网络读事件，则从网卡读取数据，并记入请求消息的缓冲中。读取完毕，则开始按照配置的协议进行解析，解析成功后，就将请求 msg 放入到 client\_conn 的 out 队列中。接下来，就对解析的命令 key 进行 hash 计算，并根据分布算法，找到对应 server 分片的连接，即一个 server\_conn 结构体，如下图。

如果 server\_conn 的 in 队列为空，首先对 server\_conn 触发一个写事件。然后将 req msg 存入到 server\_conn 的 in 队列。Server\_conn 在处理写事件时，会对 in 队列中的 req msg 进行聚合，按照 pipeline 的方式批量发送到后端资源。待发送完毕后，将该条请求 msg 从 server\_conn 的 in 队列删除，并插入到 out 队列中。

后端资源服务完成请求后，会将响应发送给 Twemproxy。当响应到 Twemproxy 后，对应的 server\_conn 会收到 epoll 读事件，则开始读取响应 msg。响应读取并解析后，会首先将 server\_conn 中，out 队列的第一个 req msg 删除，并将这个 req msg 和最新收到的 rsp msg 进行配对。在 req 和 rsp 匹配后，触发 client\_conn 的写事件，如下图。

然后 client\_conn 在处理 epoll 写事件时，则按照请求顺序，批量将响应发送给 client 端。发送完毕后，将 req msg 从 client 的 out 队列删除。最后，再回收消息缓冲，以及消息结构体，供后续请求处理的时候复用。至此一个请求的处理彻底完成。

### Twemproxy 安装和使用

Twemproxy 的安装和使用比较简单。首先通过 Git，将 Twemproxy 从 GitHub clone 到目标服务器，然后进入 Twemproxy 路径，首先执行 `$ autoreconf -fvi`，然后执行 `./configure`，最后执行 `make`（当然，也可以再执行 `make install`），这样就完成了 Twemproxy 的编译和安装。然后就可以通过 `src/nutcracker -c /xxx/conf/nutcracker.yml` 来启动 Twemproxy 了。

Twemproxy 代理后端资源访问，这些后端资源的部署信息及访问策略都是在 YAML 文件中配置。所以接下来，我们简单看一下 Twemproxy 的配置。如图所示，这个配置中代理了 2 个业务数据的缓存访问。一个是 alpha，另一个是 beta。在每个业务的配置详情里。首先是 listen 配置项，用于设置监听该业务的端口。然后是 hash 算法和分布算法。Auto\_eject\_hosts 用于设置在后端 server 异常时，是否将这个异常 server 剔除，然后进行 rehash，默认不剔除。Redis 配置项用于指示后端资源类型，是 Redis 还是 Memcached。最后一个配置项 servers，用于设置资源池列表。

以 Memcached 访问为例，将业务的 Memcached 资源部署好之后，然后将 Mc 资源列表、访问方式等设到 YAML 文件的配置项，然后启动 Twemproxy，业务端就可以通过访问 Twemproxy，来获取后端资源的数据了。后续，Mc 资源有任何变更，业务都不用做任何改变，运维直接修改 Twemproxy 的配置即可。

Twemproxy 在实际线的使用中，还是存在不少问题的。首先，它是单进程/单线程模型，一个 event\_base 要处理所有的事件，这些事件包括 client 请求的读入，转发请求给后端 server，从 server 接受响应，以及将响应发送给 client。单个 Twemproxy 实例，压测最大可以到 8w 左右的 QPS，出于线上稳定性考虑，QPS 最多支撑到 3~4w。而 Memcached 的线上 QPS，一般可以达到 10~20w，一个 Mc 实例前面要挂 3~5 个 Twemproxy 实例。实例数太多，就会引发诸如管理复杂、成本过高等一系列问题。

其次，基于性能及预防单点故障的考虑，Twemproxy 需要进行多实例部署，而且还需要根据业务访问量的变化，进行新实例的加入或冗余实例的下线。多个 Twemproxy 实例同时被访问，如果 client 访问策略不当，就会出现有些 Twemproxy 压力过大，而有些却很空闲，造成访问不均的问题。

再次，后端资源在 Twemproxy 的 YAML 文件集中配置，资源变更的维护，比直接在所有业务 client 端维护，有了很大的简化。但在多个 Twemproxy 修改配置，让这些配置同时生效，也是一个复杂的工作。

最后，Twemproxy 也无法支持 Mc 多副本、多层次架构的访问策略，无法支持 Redis 的 Master-Slave 架构的读写分离访问。

为此，你可以对 Twemproxy 进行扩展，以更好得满足业务及运维的需要。

#### Twemproxy 扩展

##### 多进程改造

性能首当其冲。首先可以对 Twemproxy 的单进程/单线程动刀，改为并行处理模型。并行方案可以用多线程方案，也可以采用多进程方案。由于 Twemproxy 只是一个消息路由中间件，不需要额外共享数据，采用多进程方案会更简洁，更适合。

多进程改造中，可以分别构建一个 master 进程和多个 worker 进程来进行任务处理，如下图所示。每个进程维护自己独立的 epoll 事件驱动。其中 master 进程，主要用于监听端口，accept 新连接，并将连接调度给 worker 进程。

而 worker 进程，基于自己独立的 event\_base，管理从 master 调度给自己的所有 client 连接。在 client 发送网络请求到达时，进行命令读取、解析，并在进程内的 IO 队列流转，最后将请求打包，pipeline 给后端的 server。

在 server 处理完毕请求，发回响应时。对应 worker 进程，会读取并解析响应，然后批量回复给 client。

通过多进程改造，Twemproxy 的 QPS 可以从 8w 提升到 40w+。业务访问时，需要部署的 Twemproxy 的实例数会大幅减少，运维会更加简洁。

##### 增加负载均衡

对于多个 Twemproxy 访问，如何进行负载均衡的问题。一般有三种方案。

第一种方案，是在 Twemproxy 和业务访问端之间，再增加一组 LVS，作为负载均衡层，通过 LVS 负载均衡层，你可以方便得增加或减少 Twemproxy 实例，由 LVS 负责负载均衡和请求分发，如下图。

第二种方案，是将 Twemproxy 的 IP 列表加入 DNS。业务 client 通过域名来访问 Twemproxy，每次建连时，DNS 随机返回一个 IP，让连接尽可能均衡。

第三种方案，是业务 client 自定义均衡策略。业务 client 从配置中心或 DNS 获取所有的 Twemproxy 的 IP 列表，然后对这些 Twemproxy 进行均衡访问，从而达到负载均衡。

方案一，可以通过成熟的 LVS 方案，高效稳定的支持负载均衡策略，但多了一层，成本和运维的复杂度会有所增加。方案二，只能做到连接均衡，访问请求是否均衡，无法保障。方案三，成本最低，性能也比前面 2 个方案更高效。推荐使用方案三，微博内部也是采用第三种方案。

增加配置中心

对于 Twemproxy 配置的维护，可以通过增加一个配置中心服务来解决。将 YAML 配置文件中的所有配置信息，包括后端资源的部署信息、访问信息，以配置的方式存储到配置中心，如下图。

Twemproxy 启动时，首先到配置中心订阅并拉取配置，然后解析并正常启动。Twemproxy 将自己的 IP 和监听端口信息，也注册到配置中心。业务 client 从配置中心，获取 Twemproxy 的部署信息，然后进行均衡访问。

在后端资源变更时，直接更新配置中心的配置。配置中心会通知所有 Twemproxy 实例，收到事件通知，Twemproxy 即可拉取最新配置，并调整后端资源的访问，实现在线变更。整个过程自动完成，更加高效和可靠。

支持 M-S-L1 多层访问

前面提到，为了应对突发洪水流量，避免硬件局部故障的影响，对 Mc 访问采用了 Master-Slave-L1 架构。可以将该缓存架构体系的访问策略，封装到 Twemproxy 内部。实现方案也比较简单。首先在 servers 配置中，增加 Master、Slave、L1 三层，如下图。

Twemproxy 启动时，每个 worker 进程预连所有的 Mc 后端，当收到 client 请求时，根据解析出来的指令，分别采用不同访问策略即可。

对于 get 请求，首先随机选择一个 L1 来访问，如果 miss，继续访问 Master 和 Slave。中间在任何一层命中，则回写。

对于 gets 请求，需要以 master 为准，从 master 读取。如果 master 获取失败，则从 slave 获取，获取后回种到 master，然后再次从 master 获取，确保得到 cas unique id 来自 master。

对于 add/cas 等请求，首先请求 master，成功后，再将 key/value 通过 set 指令，写到 slave 和所有 L1。

对于 set 请求，最简单，直接 set 所有资源池即可。

对于 stats 指令的响应，由 Twemproxy 自己统计，或者到后端 Mc 获取后聚合获得。

Redis 主从访问



Redis 支持主从复制，为了支持更大并发访问量，同时减少主库的压力，一般会部署多个从库，写操作直接请求 Redis 主库，读操作随机选择一个 Redis 从库。这个逻辑同样可以封装在Twemproxy 中。如下图所示，Redis 的主从配置信息，可以用域名的方式，也可以用 IP 端口的方式记录在配置中心，由 Twemproxy 订阅并实时更新，从而在 Redis 增减 slave、主从切换时，及时对后端进行访问变更。

本课时，讲解了大数据时代下大中型互联网系统的特点，访问 Memcached 缓存时的经典问题及应对方案；还讲解了如何通过分拆缓存池、Master-Slave 双层架构，来解决 Memcached 的容量问题、性能瓶颈、连接瓶颈、局部故障的问题，以及 Master-Slave-L1 三层架构，通过多层、多副本 Memcached 体系，来更好得解决突发洪峰流量和局部故障的问题。

本节课重点学习了基于 Twemproxy 的应用系统架构方案，学习了 Twemproxy 的系统架构和关键技术，学习了 Twemproxy 的部署及配置信息。最后还学习了如何扩展 Twemproxy，从而使 Twemproxy 具有更好的性能、可用性和可运维性。

可以参考下面的思维导图，对这些知识点进行回顾和梳理。

OK，这节课就讲到这里啦，下一课时我将分享“Redis基本原理”，记得按时来听课哈。好，下节课见，拜拜！

## 第六章：Redis原理、协议及使用

---

### 第16讲：常用的缓存组件Redis是如何运行的？

你好，我是你的缓存课老师陈波，欢迎进入第 16 课时“Redis 基本原理”的学习。

Redis 基本原理

Redis 简介

Redis 是一款基于 ANSI C 语言编写的，BSD 许可的，日志型 key-value 存储组件，它的所有数据结构都存在内存中，可以用作缓存、数据库和消息中间件。

Redis 是 Remote dictionary server 即远程字典服务的缩写，一个 Redis 实例可以有多个存储数据的字典，客户端可以通过 select 来选择字典即 DB 进行数据存储。

Redis 特性

同为 key-value 存储组件，Memcached 只能支持二进制字节块这一种数据类型。而 Redis 的数据类型却丰富的多，它具有 8 种核心数据类型，每种数据类型都有一系列操作指令对应。Redis 性能很高，单线程压测可以达到 10~11w 的 QPS。

虽然 Redis 所有数据的读写操作，都在内存中进行，但也可以将所有数据进行落盘做持久化。Redis 提供了 2 种持久化方式。

快照方式，将某时刻所有数据都写入硬盘的 RDB 文件；

追加文件方式，即将所有写命令都以追加的方式写入硬盘的 AOF 文件中。

线上 Redis 一般会同时使用两种方式，通过开启 appendonly 及关联配置项，将写命令及时追加到 AOF 文件，同时在每日流量低峰时，通过 bgsave 保存当时所有内存数据快照。

对于互联网系统的线上流量，读操作远远大于写操作。以微博为例，读请求占总体流量的 90% 左右。大量的读请求，通常会远超 Redis 的可承载范围。此时，可以使用 Redis 的复制特性，让一个 Redis 实例作为 master，然后通过复制挂载多个不断同步更新的副本，即多个 slave。通过读写分离，把所有写操作落在 Redis 的 master，所有读操作随机落在 Redis 的多个 slave 中，从而大幅提升 Redis 的读写能力。

Lua 是一个高效、简洁、易扩展的脚本语言，可以方便的嵌入其他语言中使用。Redis 自 2.6 版本开始支持 Lua。通过支持 client 端自定义的 Lua 脚本，Redis 可以减少网络开销，提升处理性能，还可以把脚本中的多个操作作为一个整体来操作，实现原子性更新。

Redis 还支持事务，在 multi 指令后，指定多个操作，然后通过 exec 指令一次性执行，中途如果出现异常，则不执行所有命令操作，否则，按顺序一次性执行所有操作，执行过程中不会执行任何其他指令。

Redis 还支持 Cluster 特性，可以通过自动或手动方式，将所有 key 按哈希分散到不同节点，在容量不足时，还可以通过 Redis 的迁移指令，把其中一部分 key 迁移到其他节点。

对于 Redis 的特性，可以通过这张思维导图，做个初步了解。在后面的课程中，我会逐一进行详细讲解。

作为缓存组件，Redis 的最大优势是支持丰富的数据类型。目前，Redis 支持 8 种核心数据类型，包括 string、list、set、sorted set、hash、bitmap、geo、hyperloglog。

Redis 的所有内存数据结构都存在全局的 dict 字典中，dict 类似 Memcached 的 hashtable。Redis 的 dict 也有 2 个哈希表，插入新 key 时，一般用 0 号哈希表，随着 key 的插入或删除，当 0 号哈希表的 keys 数大于哈希表桶数，或 keys 数小于哈希桶的 1/10 时，就对 hash 表进行扩缩。dict 中，哈希表解决冲突的方式，与 Memcached 相同，也是使用桶内单链表，来指向多个 hash 相同的 key/value 数据。

## Redis 高性能

Redis 一般被看作单进程/单线程组件，因为 Redis 的网络 IO 和命令处理，都在核心进程中由单线程处理。Redis 基于 Epoll 事件模型开发，可以进行非阻塞网络 IO，同时由于单线程命令处理，整个处理过程不存在竞争，不需要加锁，没有上下文切换开销，所有数据操作都是在内存中操作，所以 Redis 的性能很高，单个实例即可以达到 10w 级的 QPS。核心线程除了负责网络 IO 及命令处理外，还负责写数据到缓冲，以方便将最新写操作同步到 AOF、slave。

除了主进程，Redis 还会 fork 一个子进程，来进行重负荷任务的处理。Redis fork 子进程主要有 3 种场景。

收到 `bgrewriteaof` 命令时，Redis 调用 `fork`，构建一个子进程，子进程往临时 AOF 文件中，写入重建数据库状态的所有命令，当写入完毕，子进程则通知父进程，父进程把新增的写操作也追加到临时 AOF 文件，然后将临时文件替换老的 AOF 文件，并重命名。

收到 `bgsave` 命令时，Redis 构建子进程，子进程将内存中的所有数据通过快照做一次持久化落地，写入到 RDB 中。

当需要进行全量复制时，master 也会启动一个子进程，子进程将数据库快照保存到 RDB 文件，在写完 RDB 快照文件后，master 就会把 RDB 发给 slave，同时将后续新的写指令都同步给 slave。

主进程中，除了主线程处理网络 IO 和命令操作外，还有 3 个辅助 BIO 线程。这 3 个 BIO 线程分别负责处理，文件关闭、AOF 缓冲数据刷新到磁盘，以及清理对象这三个任务队列。

Redis 在启动时，会同时启动这三个 BIO 线程，然后 BIO 线程休眠等待任务。当需要执行相关类型的后台任务时，就会构建一个 `bio_job` 结构，记录任务参数，然后将 `bio_job` 追加到任务队列尾部。然后唤醒 BIO 线程，即可进行任务执行。

#### Redis 持久化

Redis 的持久化是通过 RDB 和 AOF 文件进行的。RDB 只记录某个时间点的快照，可以通过设置指定时间内修改 keys 数的阈值，超过则自动构建 RDB 内容快照，不过线上运维，一般会选择在业务低峰期定期进行。RDB 存储的是构建时刻的数据快照，内存数据一旦落地，不会理会后续的变更。而 AOF，记录是构建整个数据库内容的命令，它会随着新的写操作不断进行追加操作。由于不断追加，AOF 会记录数据大量的中间状态，AOF 文件会变得非常大，此时，可以通过 `bgrewriteaof` 指令，对 AOF 进行重写，只保留数据的最后内容，来大大缩减 AOF 的内容。

为了提升系统的可扩展性，提升读操作的支撑能力，Redis 支持 master-slave 的复制功能。当 Redis 的 slave 部署并设置完毕后，slave 会和 master 建立连接，进行全量同步。

第一次建立连接，或者长时间断开连接后，缺失的指令超过 master 复制缓冲区的大小，都需要先进行一次全量同步。全量同步时，master 会启动一个子进程，将数据库快照保存到文件中，然后将这个快照文件发给 slave，同时将快照之后的写指令也同步给 slave。

全量同步完成后，如果 slave 短时间中断，然后重连复制，缺少的写指令长度小于 master 的复制缓冲大小，master 就会把 slave 缺失的内容全部发送给 slave，进行增量复制。

Redis 的 master 可以挂载多个 slave，同时 slave 还可以继续挂载 slave，通过这种方式，可以有效减轻 master 的压力，同时在 master 挂掉后，可以在 slave 通过 `slaveof no one` 指令，使当前 slave 停止与 master 的同步，转而成为新的 master。

### Redis 集群管理

Redis 的集群管理有 3 种方式。

`client` 分片访问，`client` 对 `key` 做 `hash`，然后按取模或一致性 `hash`，把 `key` 的读写分散到不同的 Redis 实例上。

在 Redis 前加一个 `proxy`，把路由策略、后端 Redis 状态维护的工作都放到 `proxy` 中进行，`client` 直接访问 `proxy`，后端 Redis 变更，只需修改 `proxy` 配置即可。

直接使用 Redis `cluster`。Redis 创建之初，使用方直接给 Redis 的节点分配 `slot`，后续访问时，对 `key` 做 `hash` 找到对应的 `slot`，然后访问 `slot` 所在的 Redis 实例。在需要扩容缩容时，可以在线通过 `cluster setslot` 指令，以及 `migrate` 指令，将 `slot` 下所有 `key` 迁移到目标节点，即可实现扩缩容的目的。

至此，Redis 的基本原理就讲完了，相信你对 Redis 应该有了一个大概的了解。接下来，我将开始逐一深入分析 Redis 的各个技术细节。

OK，这节课就讲到这里啦，下一课时我将分享“Redis 数据类型”，记得按时来听课哈。好，下节课见，拜拜！

## 第17讲：如何理解、选择并使用Redis的核心数据类型？

你好，我是你的缓存课老师陈波，欢迎进入第 17 课时“Redis 数据类型”的学习。

### Redis 数据类型

首先，来看一下 Redis 的核心数据类型。Redis 有 8 种核心数据类型，分别是：

`string` 字符串类型；

`list` 列表类型；

`set` 集合类型；

`sorted set` 有序集合类型；

`hash` 类型；

`bitmap` 位图类型；

`geo` 地理位置类型；

`HyperLogLog` 基数统计类型。

---

## string 字符串

string 是 Redis 的最基本数据类型。可以把它理解为 Mc 中 key 对应的 value 类型。string 类型是二进制安全的，即 string 中可以包含任何数据。

Redis 中的普通 string 采用 raw encoding 即原始编码方式，该编码方式会动态扩容，并通过提前预分配冗余空间，来减少内存频繁分配的开销。

在字符串长度小于 1MB 时，按所需长度的 2 倍来分配，超过 1MB，则按照每次额外增加 1MB 的容量来预分配。

Redis 中的数字也存为 string 类型，但编码方式跟普通 string 不同，数字采用整型编码，字符串内容直接设为整数值的二进制字节序列。

在存储普通字符串，序列化对象，以及计数器等场景时，都可以使用 Redis 的字符串类型，字符串数据类型对应使用的指令包括 set、get、mset、incr、decr 等。

## list 列表

Redis 的 list 列表，是一个快速双向链表，存储了一系列的 string 类型的字符串值。list 中的元素按照插入顺序排列。插入元素的方式，可以通过 lpush 将一个或多个元素插入到列表的头部，也可以通过 rpush 将一个或多个元素插入到队列尾部，还可以通过 lset、linsert 将元素插入到指定位置或指定元素的前后。

list 列表的获取，可以通过 lpop、rpop 从对头或队尾弹出元素，如果队列为空，则返回 nil。还可以通过 Blpop、Brpop 从队头/队尾阻塞式弹出元素，如果 list 列表为空，没有元素可供弹出，则持续阻塞，直到有其他 client 插入新的元素。这里阻塞弹出元素，可以设置过期时间，避免无限期等待。最后，list 列表还可以通过 LrangeR 获取队列内指定范围内的所有元素。Redis 中，list 列表的偏移位置都是基于 0 的下标，即列表第一个元素的下标是 0，第二个是 1。偏移量也可以是负数，倒数第一个是 -1，倒数第二个是 -2，依次类推。

list 列表，对于常规的 pop、push 元素，性能很高，时间复杂度为  $O(1)$ ，因为是列表直接追加或弹出。但对于通过随机插入、随机删除，以及随机范围获取，需要轮询列表确定位置，性能就比较低下了。

feed timeline 存储时，由于 feed id 一般是递增的，可以直接存为 list，用户发表新 feed，就直接追加到队尾。另外消息队列、热门 feed 等业务场景，都可以使用 list 数据结构。

操作 list 列表时，可以用 lpush、lpop、rpush、rpop、lrange 来进行常规的队列进出及范围获取操作，在某些特殊场景下，也可以用 lset、linsert 进行随机插入操作，用 lrem 进行指定元素删除操作；最后，在消息列表的消费时，还可以用 Blpop、Brpop 进行阻塞式获取，从而在列表暂时没有元素时，可以安静的等待新元素的插入，而不需要额外持续的查询。

## set 集合

set 是 string 类型的无序集合，set 中的元素是唯一的，即 set 中不会出现重复的元素。Redis 中的集合一般是通过 dict 哈希表实现的，所以插入、删除，以及查询元素，可以根据元素 hash 值直接定位，时间复杂度为  $O(1)$ 。

对 set 类型数据的操作，除了常规的添加、删除、查找元素外，还可以用以下指令对 set 进行操作。

`sismember` 指令判断该 key 对应的 set 数据结构中，是否存在某个元素，如果存在返回 1，否则返回 0；

`sdiff` 指令来对多个 set 集合执行差集；

`sinter` 指令对多个集合执行交集；

`sunion` 指令对多个集合执行并集；

`spop` 指令弹出一个随机元素；

`srandmember` 指令返回一个或多个随机元素。

set 集合的特点是查找、插入、删除特别高效，时间复杂度为  $O(1)$ ，所以在社交系统中，可以用于存储关注的好友列表，用来判断是否关注，还可以用来做好友推荐使用。另外，还可以利用 set 的唯一性，来对服务的来源业务、来源 IP 进行精确统计。

## sorted set 有序集合

Redis 中的 sorted set 有序集合也称为 zset，有序集合同 set 集合类似，也是 string 类型元素的集合，且所有元素不允许重复。

但有序集合中，每个元素都会关联一个 double 类型的 score 分数值。有序集合通过这个 score 值进行由小到大的排序。有序集合中，元素不允许重复，但 score 分数值却允许重复。

有序集合除了常规的添加、删除、查找元素外，还可以通过以下指令对 sorted set 进行操作。

`zscan` 指令：按顺序获取有序集合中的元素；

`zscore` 指令：获取元素的 score 值；

`zrange` 指令：通过指定 score 返回获取 score 范围内的元素；

在某个元素的 score 值发生变更时，还可以通过 `zincrby` 指令对该元素的 score 值进行加减。

通过 `zinterstore`、`zunionstore` 指令对多个有序集合进行取交集和并集，然后将新的有序集合存到一个新的 key 中，如果有重复元素，重复元素的 score 进行相加，然后作为新集合中该元素的 score 值。

sorted set 有序集合的特点是：

所有元素按 `score` 排序，而且不重复；

查找、插入、删除非常高效，时间复杂度为  $O(1)$ 。

因此，可以用有序集合来统计排行榜，实时刷新榜单，还可以用来记录学生成绩，从而轻松获取某个成绩范围内的学生名单，还可以用来对系统统计增加权重值，从而在 dashboard 实时展示。

hash 哈希

Redis 中的哈希实际是 field 和 value 的一个映射表。

hash 数据结构的特点是在单个 key 对应的哈希结构内部，可以记录多个键值对，即 field 和 value 对，value 可以是任何字符串。而且这些键值对查询和修改很高效。

所以可以用 hash 来存储具有多个元素的复杂对象，然后分别修改或获取这些元素。hash 结构中的一些重要指令，包括：`hmset`、`hmget`、`hexists`、`hgetall`、`hincrby` 等。

`hmset` 指令批量插入多个 field、value 映射；

`hmget` 指令获取多个 field 对应的 value 值；

`hexists` 指令判断某个 field 是否存在；

如果 field 对应的 value 是整数，还可以用 `hincrby` 来对该 value 进行修改。

bitmap 位图

Redis 中的 bitmap 位图是一串连续的二进制数字，底层实际是基于 string 进行封装存储的，按 bit 位进行指令操作的。bitmap 中每一 bit 位所在的位置就是 offset 偏移，可以用 `setbit`、`bitfield` 对 bitmap 中每个 bit 进行置 0 或置 1 操作，也可以用 `bitcount` 来统计 bitmap 中的被置 1 的 bit 数，还可以用 `bitop` 来对多个 bitmap 进行求与、或、异或等操作。

bitmap 位图的特点是按位设置、求与、求或等操作很高效，而且存储成本非常低，用来存对象标签属性的话，一个 bit 即可存一个标签。可以用 bitmap，存用户最近 N 天的登录情况，每天用 1 bit，登录则置 1。个性推荐在社交应用中非常重要，可以对新闻、feed 设置一系列标签，如军事、娱乐、视频、图片、文字等，用 bitmap 来存储这些标签，在对应标签 bit 位上置 1。对用户，也可以采用类似方式，记录用户的多种属性，并可以很方便的根据标签来进行多维度统计。bitmap 位图的重要指令包括：`setbit`、`getbit`、`bitcount`、`bitfield`、`bitop`、`bitpos` 等。

在移动社交时代，LBS 应用越来越多，比如微信、陌陌中附近的人，美团、大众点评中附近的美食、电影院，滴滴、优步中附近的专车等。要实现这些功能，就得使用地理位置信息进行搜索。地球的地理位置是使用二维的经纬度进行表示的，我们只要确定一个点的经纬度，就可以确认它在地球的位置。

Redis 在 3.2 版本之后增加了对 GEO 地理位置的处理功能。Redis 的 GEO 地理位置本质上是基于 sorted set 封装实现的。在存储分类 key 下的地理位置信息时，需要对该分类 key 构建一个 sorted set 作为内部存储结构，用于存储一系列位置点。



在存储某个位置点时，首先利用 Geohash 算法，将该位置二维的经纬度，映射编码成一维的 52 位整数值，将位置名称、经纬度编码 score 作为键值对，存储到分类 key 对应的 sorted set 中。

需要计算某个位置点 A 附近的人时，首先以指定位置 A 为中心点，以距离作为半径，算出 GEO 哈希 8 个方位的范围，然后依次轮询方位范围内的所有位置点，只要这些位置点到中心位置 A 的距离在要求距离范围内，就是目标位置点。轮询完所有范围内的位置点后，重新排序即得到位置点 A 附近的所有目标。

使用 `geoadd`，将位置名称（如人、车辆、店名）与对应的地理位置信息添加到指定的位置分类 key 中；

使用 `geopos` 方便地查询某个名称所在的位置信息；

使用 `georadius` 获取指定位置附近，不超过指定距离的所有元素；

使用 `geodist` 来获取指定的两个位置之间的距离。

这样，是不是就可以实现，找到附近的餐厅，算出当前位置到对应餐厅的距离，这样的功能了？

Redis GEO 地理位置，利用 Geohash 将大量的二维经纬度转一维的整数值，这样可以方便的对地理位置进行查询、距离测量、范围搜索。但由于地理位置点非常多，一个地理分类 key 下可能会有大量元素，在 GEO 设计时，需要提前进行规划，避免单 key 过度膨胀。

Redis 的 GEO 地理位置数据结构，应用场景很多，比如查询某个地方的具体位置，查当前位置到目的地的距离，查附近的人、餐厅、电影院等。GEO 地理位置数据结构中，重要指令包括 `geoadd`、`geopos`、`geodist`、`georadius`、`georadiusbymember` 等。

hyperLogLog 基数统计

Redis 的 hyperLogLog 是用来做基数统计的数据类型，当输入巨大数量的元素做统计时，只需要很小的内存即可完成。HyperLogLog 不保存元数据，只记录待统计元素的估算数量，这个估算数量是一个带有 0.81% 标准差的近似值，在大多数业务场景，对海量数据，不足 1% 的误差是可以接受的。

Redis 的 HyperLogLog 在统计时，如果计数数量不大，采用稀疏矩阵存储，随着计数的增加，稀疏矩阵占用的空间也会逐渐增加，当超过阈值后，则改为稠密矩阵，稠密矩阵占用的空间是固定的，约为 12KB 字节。

通过 hyperLoglog 数据类型，你可以利用 `pfadd` 向基数统计中增加新的元素，可以用 `pfcount` 获得 hyperLogLog 结构中存储的近似基数数量，还可以用 `hypermerge` 将多个 hyperLogLog 合并为一个 hyperLogLog 结构，从而可以方便的获取合并后的基数数量。

hyperLogLog 的特点是统计过程不记录独立元素，占用内存非常少，非常适合统计海量数据。在大中型系统中，统计每日、每月的 UV 即独立访客数，或者统计海量用户搜索的独立词条数，都可以用 hyperLogLog 数据类型来进行处理。

OK，这节课就讲到这里啦，下一课时我将分享“Redis 协议分析”，记得按时来听课哈。好，下节课见，拜拜！

第18讲：Redis协议的请求和响应有哪些“套路”可循？

2019/10/14 陈波

8.3M

00:00/12:01

看视频

你好，我是你的缓存课老师陈波，欢迎进入第 18 课时“Redis 协议分析”的学习，本课时主要学习Redis的设计原则、三种响应模式、2种请求格式、5种响应格式。

Redis 协议

Redis 支持 8 种核心数据结构，每种数据结构都有一系列的操作指令，除此之外，Redis 还有事务、集群、发布订阅、脚本等一系列相关的指令。为了方便以一种统一的风格和原则来设计和使用这些指令，Redis 设计了 RESP，即 Redis Serialization Protocol，中文意思是 Redis 序列化协议。RESP 是二进制安全协议，可以供 Redis 或其他任何 Client-Server 使用。在 Redis 内部，还会基于 RESP 进一步扩展细节。

设计原则

Redis 序列化协议的设计原则有三个：

第一是实现简单；

第二是可快速解析；

第三是便于阅读。

Redis 协议的请求响应模型有三种，除了 2 种特殊模式，其他基本都是 ping-pong 模式，即 client 发送一个请求，server 回复一个响应，一问一答的访问模式。

2 种特殊模式：

pipeline 模式，即 client 一次连续发送多个请求，然后等待 server 响应，server 处理完请求后，把响应返回给 client。

pub/sub 模式。即发布订阅模式，client 通过 subscribe 订阅一个 channel，然后 client 进入订阅状态，静静等待。当有消息产生时，server 会持续自动推送消息给 client，不需要 client 的额外请求。而且客户端在进入订阅状态后，只可接受订阅相关的命令如 SUBSCRIBE、PSUBSCRIBE、UNSUBSCRIBE 和 PUNSUBSCRIBE，除了这些命令，其他命令一律失效。

Redis 协议的请求和响应也是有固定套路的。

对于请求指令，格式有 2 种类型。

当你没有 `redis-client`，但希望可以用通用工具 `telnet`，直接与 Redis 交互时，Redis 协议虽然简单易于阅读，但在交互式会话中使用，并不容易拼写，此时可以用第一种格式，即 `inline cmd` 内联命令格式。使用 `inline cmd` 内联格式，只需要用空格分隔请求指令及参数，简单快速，一个简单的例子如 `mget key1 key2\r\n`。

第二种格式是 `Array` 数组格式类型。请求指令用的数组类型，与 Redis 响应的数组类型相同，后面在介绍响应格式类型时会详细介绍。

## 响应格式

Redis 协议的响应格式有 5 种，分别是：

`simple strings` 简单字符串类型，以 `+` 开头，后面跟字符串，以 `CRLF`（即 `\r\n`）结尾。这种类型不是二进制安全类型，字符串中不能包含 `\r` 或者 `\n`。比如许多响应回复以 `OK` 作为操作成功的标志，协议内容就是 `+OK\r\n`。

Redis 协议将错误作为一种专门的类型，格式同简单字符串类型，唯一不同的是以 `-`（减号）开头。Redis 内部实现对 Redis 协议做了进一步规范，减号后面一般先跟 `ERR` 或者 `WRONGTYPE`，然后再跟其他简单字符串，最后以 `CRLF`（回车换行）结束。这里给了两个示例，`client` 在解析响应时，一旦发现 `-` 开头，就知道收到 `Error` 响应。

`Integer` 整数类型。整数类型以 `:` 开头，后面跟字符串表示的数字，最后以回车换行结尾。Redis 中许多命令都返回整数，但整数的含义要由具体命令来确定。比如，对于 `incr` 指令，`:` 后的整数表示变更后的数值；对于 `llen` 表示 `list` 列表的长度，对于 `exists` 指令，`1` 表示 `key` 存在，`0` 表示 `key` 不存在。这里给个例子，`:` 后面跟了个 `1000`，然后回车换行结束。

`bulk strings` 字符串块类型。字符串块分头部和真正字符串内容两部分。字符串块类型的头部，以 `$` 开头，随后跟真正字符串内容的字节长度，然后以 `CRLF` 结尾。字符串块的头部之后，跟随真正的字符串内容，最后以 `CRLF` 结束字符串块。字符串块用于表示二进制安全的字符串，最大长度可以支持 `512MB`。一个常规的例子，`"$6\r\nfoobar\r\n"`，对于空字符串，可以表示为 `"$0\r\n\r\n"`，`NULL` 字符串：`"$-1\r\n"`。

`Arrays` 数组类型，如果一个命令需要返回多条数据就需要用数组格式类型，另外，前面提到 `client` 的请求命令也是主要采用这种格式。

`Arrays` 数组类型，以 `*` 开头，随后跟一个数组长度 `N`，然后以回车换行结尾；然后后面跟随 `N` 个数组元素，每个数组元素的类型，可以是 Redis 协议中除内联格式外的任何一种类型。

比如一个字符串块的数组实例，`2\r\n$3\r\nget\r\n$3\r\nkey\r\n`。整数数组实

例：`"3\r\n:1\r\n:2\r\n:3\r\n"`，混合数组实例：`"3\r\n:1\r\n-Bar\r\n$6\r\nfoobar\r\n"`，空数组：`"0\r\n"`，`NULL`数组：`"*-1\r\n"`。

协议分类

Redis 协议主要分为 16 种，其中 8 种协议对应前面我们讲到的 8 种数据类型，你选择了使用什么数据类型，就使用对应的响应操作指令即可。剩下 8 种协议如下所示。

pub-sub 发布订阅协议，client 可以订阅 channel，持续等待 server 推送消息。

事务协议，事务协议可以用 multi 和 exec 封装一些列指令，来一次性执行。

脚本协议，关键指令是 eval、evalsha 和 script 等。

连接协议，主要包括权限控制，切换 DB，关闭连接等。

复制协议，包括 slaveof、role、psync 等。

配置协议，config set/get 等，可以在线修改/获取配置。

调试统计协议，如 slowlog, monitor, info 等。

其他内部命令，如 migrate, dump, restore 等。

## Redis client 的使用及改进

由于 Redis 使用广泛，几乎所有主流语言都有对 Redis 开发了对应的 client。以 Java 语言为例，广泛使用的有 Jedis、Redisson 等。对于 Jedis client，它的优势是轻量，简洁，便于集成和改造，它支持连接池，提供指令维度的操作，几乎支持 Redis 的所有指令，但它不支持读写分离。Redisson 基于 Netty 实现，非阻塞 IO，性能较高，而且支持异步请求和连接池，还支持读写分离、读负载均衡，它内建了 tomcat Session，支持 spring session 集成，但 redisson 实现相对复杂。

在新项目启动时，如果只是简单的 Redis 访问业务场景，可以直接用 Jedis，甚至可以简单封装 Jedis，实现 master-slave 的读写分离方案。如果想直接使用读写分离，想集成 spring session 等这些高级特性，也可以采用 redisson。

Redis client 在使用中，需要根据业务及运维的需要，进行相关改进。在 client 访问异常时，可以增加重试策略，在访问某个 slave 异常时，需要重试其他 slave 节点。需要增加对 Redis 主从切换、slave 扩展的支持，比如采用守护线程定期扫描 master、slave 域名，发现 IP 变更，及时切换连接。对于多个 slave 的访问，还需要增加负载均衡策略。最后，Redis client 还可以与配置中心、Redis 集群管理平台整合，从而实时感知及协调 Redis 服务的访问。

至此，本节课的内容就讲完了。

在这几节课中，你首先学习了 Redis 的特性及基本原理，初步了解了 Redis 的数据类型、主进程/子进程、BIO 线程、持久化、复制、集群等；这些内容会在后续逐一深入学习。

然后，详细学习了 Redis 的数据类型，了解了字符串、列表、集合、有序集合、哈希、位图、GEO 地理位置、HyperLogLog 基数统计，这 8 种核心数据类型的功能、特点、主要操作指令及应用场景。

接下来，你还熟悉了 Redis 协议，包括 Redis 协议的设计原则、三种响应模型，2 种请求格式和 5 种响应格式。

最后，以 Java 语言为例，你还了解了 Redis client 的对比、选择及改进。

你可以参考这个思维导图，对这些知识点进行回顾和梳理。

OK，这节课就讲到这里啦，下一课时我将分享“Redis 系统架构”，记得按时来听课哈。好，下节课见，拜拜！

## 第七章：Redis进阶（上）

---

### 第19讲：Redis系统架构中各个处理模块是干什么的？

你好，我是你的缓存课老师陈波，欢迎进入第 19 课时“Redis 系统架构”的学习。

Redis 系统架构

通过前面的学习，相信你已经掌握了 Redis 的原理、数据类型及访问协议等内容。本课时，我将进一步分析 Redis 的系统架构，重点讲解 Redis 系统架构的事件处理机制、数据管理、功能扩展、系统扩展等内容。

事件处理机制

Redis 组件的系统架构如图所示，主要包括事件处理、数据存储及管理、用于系统扩展的主从复制/集群管理，以及为插件化功能扩展的 Module System 模块。

Redis 中的事件处理模块，采用的是作者自己开发的 ae 事件驱动模型，可以进行高效的网络 IO 读写、命令执行，以及时间事件处理。

其中，网络 IO 读写处理采用的是 IO 多路复用技术，通过对 evport、epoll、kqueue、select 等进行封装，同时监听多个 socket，并根据 socket 目前执行的任务，来为 socket 关联不同的事件处理器。

当监听端口对应的 socket 收到连接请求后，就会创建一个 client 结构，通过 client 结构来对连接状态进行管理。在请求进入时，将请求命令读取缓冲并进行解析，并存入到 client 的参数列表。

然后根据请求命令找到对应的 redisCommand，最后根据命令协议，对请求参数进一步的解析、校验并执行。Redis 中时间事件比较简单，目前主要是执行 serverCron，来做一些统计更新、过期 key 清理、AOF 及 RDB 持久化等辅助操作。

数据管理

Redis 的内存数据都存在 redisDB 中。Redis 支持多 DB，每个 DB 都对应一个 redisDB 结构。Redis 的 8 种数据类型，每种数据类型都采用一种或多种内部数据结构进行存储。同时这些内部数据结构及数据相关的辅助信息，都以 key/value 的格式存在 redisDB 中的各个 dict 字典中。

数据在写入 redisDB 后，这些执行的写指令还会及时追加到 AOF 中，追加的方式是先实时写入 AOF 缓冲，然后按策略刷缓冲数据到文件。由于 AOF 记录每个写操作，所以一个 key 的大量中间状态也会呈现在 AOF 中，导致 AOF 冗余信息过多，因此 Redis 还设计了一个 RDB 快照操作，可以通过定期将内存里所有的数据快照落地到 RDB 文件，来以最简洁的方式记录 Redis 的所有内存数据。

Redis 进行数据读写的核心处理线程是单线程模型，为了保持整个系统的高性能，必须避免任何 kernel 导致阻塞的操作。为此，Redis 增加了 BIO 线程，来处理容易导致阻塞的文件 close、fsync 等操作，确保系统处理的性能和稳定性。

在 server 端，存储内存永远是昂贵且短缺的，Redis 中，过期的 key 需要及时清理，不活跃的 key 在内存不足时也可能需要进行淘汰。为此，Redis 设计了 8 种淘汰策略，借助新引入的 eviction pool，进行高效的 key 淘汰和内存回收。

#### 功能扩展

Redis 在 4.0 版本之后引入了 Module System 模块，可以方便使用者，在不修改核心功能的同时，进行插件化功能开发。使用者可以将新的 feature 封装成动态链接库，Redis 可以在启动时加载，也可以在运行过程中随时按需加载和启用。

在扩展模块中，开发者可以通过 RedisModule\_init 初始化新模块，用 RedisModule\_CreateCommand 扩展各种新模块指令，以可插拔的方式为 Redis 引入新的数据结构和访问命令。

#### 系统扩展

Redis 作者在架构设计中对系统的扩展也倾注了大量关注。在主从复制功能中，psync 在不断的优化，不仅在 slave 闪断重连后可以进行增量复制，而且在 slave 通过主从切换成为 master 后，其他 slave 仍然可以与新晋升的 master 进行增量复制，另外，其他一些场景，如 slave 重启后，也可以进行增量复制，大大提升了主从复制的可用性。使用者可以更方便的使用主从复制，进行业务数据的读写分离，大幅提升 Redis 系统的稳定读写能力。

通过主从复制可以较好的解决 Redis 的单机读写问题，但所有写操作都集中在 master 服务器，很容易达到 Redis 的写上限，同时 Redis 的主从节点都保存了业务的所有数据，随着业务发展，很容易出现内存不够用的问题。

为此，Redis 分区无法避免。虽然业界大多采用在 client 和 proxy 端分区，但 Redis 自己也早早推出了 cluster 功能，并不断进行优化。Redis cluster 预先设定了 16384 个 slot 槽，在 Redis 集群启动时，通过手动或自动将这些 slot 分配到不同服务节点上。在进行 key 读写定位时，首先对 key 做 hash，并将 hash 值对 16383，做按位与运算，确认 slot，然后确认服务节点，最后再对对应的 Redis 节点，进行常规读写。如果 client 发送到错误的 Redis 分片，Redis 会发送重定向回复。如果业务数据大量增加，Redis 集群可以通过数据迁移，来进行在线扩容。

OK，这节课就讲到这里啦，下一课时我将重点讲解“Redis 的事件驱动模型”，记得按时来听课哈。好，下节课见，拜拜！

## 第20讲：Redis如何处理文件事件和时间事件？

上一课时，我们学习了 Redis 的系统架构，接下来的几个课时我将带你一起对这些模块和设计进行详细分析。首先，我将分析 Redis 的事件驱动模型。

Redis 事件驱动模型

事件驱动模型

Redis 是一个事件驱动程序，但和 Memcached 不同的是，Redis 并没有采用 libevent 或 libev 这些开源库，而是直接开发了一个新的事件循环组件。Redis 作者给出的理由是，尽量减少外部依赖，而自己开发的事件模型也足够简洁、轻便、高效，也更易控制。Redis 的事件驱动模型机制封装在 aeEventLoop 等相关的结构体中，网络连接、命令读取执行回复，数据的持久化、淘汰回收 key 等，几乎所有的核心操作都通过 ae 事件模型进行处理。

Redis 的事件驱动模型处理 2 类事件：

文件事件，如连接建立、接受请求命令、发送响应等；

时间事件，如 Redis 中定期要执行的统计、key 淘汰、缓冲数据写出、rehash等。

文件事件处理

Redis 的文件事件采用典型的 Reactor 模式进行处理。Redis 文件事件处理机制分为 4 部分：

连接 socket

IO 多路复用程序

文件事件分派器

事件处理器

文件事件是对连接 socket 操作的一个抽象。当端口监听 socket 准备 accept 新连接，或者连接 socket 准备好读取请求、写入响应、关闭时，就会产生一个文件事件。IO 多路复用程序负责同时监听多个 socket，当这些 socket 产生文件事件时，就会触发事件通知，文件分派器就会感知并获取到这些事件。

虽然多个文件事件可能会并发出现，但 IO 多路复用程序总会将所有产生事件的 socket 放入一个队列中，通过这个队列，有序的把这些文件事件通知给文件分派器。

IO多路复用



Redis 封装了 4 种多路复用程序，每种封装实现都提供了相同的 API 实现。编译时，会按照性能和系统平台，选择最佳的 IO 多路复用函数作为底层实现，选择顺序是，首先尝试选择 Solaris 中的 `evport`，如果没有，就尝试选择 Linux 中的 `epoll`，否则就选择大多 UNIX 系统都支持的 `kqueue`，这 3 个多路复用函数都直接使用系统内核内部的结构，可以服务数十万的文件描述符。

如果当前编译环境没有上述函数，就会选择 `select` 作为底层实现方案。`select` 方案的性能较差，事件发生时，会扫描全部监听的描述符，事件复杂度是  $O(n)$ ，并且只能同时服务有限个文件描述符，32 位机默认是 1024 个，64 位机默认是 2048 个，所以一般情况下，并不会选择 `select` 作为线上运行方案。Redis 的这 4 种实现，分别在 `ae_evport`、`ae_epoll`、`ae_kqueue` 和 `ae_select` 这 4 个代码文件中。

### 文件事件收集及派发器

Redis 中的文件事件分发器是 `aeProcessEvents` 函数。它会首先计算最大可以等待的时间，然后利用 `aeApiPoll` 等待文件事件的发生。如果在等待时间内，一旦 IO 多路复用程序产生了事件通知，则会立即轮询所有已产生的文件事件，并将文件事件放入 `aeEventLoop` 中的 `aeFiredEvents` 结构数组中。每个 `fired event` 会记录 `socket` 及 Redis 读写事件类型。

这里会涉及将多路复用中的事件类型，转换为 Redis 的 `ae` 事件驱动模型中的事件类型。以采用 Linux 中的 `epoll` 为例，会将 `epoll` 中的 `EPOLLIN` 转为 `AE_READABLE` 类型，将 `epoll` 中的 `EPOLLOUT`、`EPOLLERR` 和 `EPOLLHUP` 转为 `AE_WRITABLE` 事件。

`aeProcessEvents` 在获取到触发的事件后，会根据事件类型，将文件事件 `dispatch` 派发给对应事件处理函数。如果同一个 `socket`，同时有读事件和写事件，Redis 派发器会首先派发处理读事件，然后再派发处理写事件。

### 文件事件处理函数分类

Redis 中文件事件函数的注册和处理主要分为 3 种。

连接处理函数 `acceptTcpHandler`

Redis 在启动时，在 `initServer` 中对监听的 `socket` 注册读事件，事件处理器为 `acceptTcpHandler`，该函数在有新连接进入时，会被派发器派发读任务。在处理该读任务时，会 `accept` 新连接，获取调用方的 IP 及端口，并对新连接创建一个 `client` 结构。如果同时有大量连接同时进入，Redis 一次最多处理 1000 个连接请求。

`readQueryFromClient` 请求处理函数

连接函数在创建 `client` 时，会对新连接 `socket` 注册一个读事件，该读事件的事件处理器就是 `readQueryFromClient`。在连接 `socket` 有请求命令到达时，IO 多路复用程序会获取并触发文件事件，然后这个读事件被派发器派发给本请求的处理函数。`readQueryFromClient` 会从连接 `socket` 读取数据，存入 `client` 的 `query` 缓冲，然后进行解析命令，按照 Redis 当前支持的 2 种请求格式，及 `inline` 内联格式和 `multibulk` 字符块数组格式进行尝试解析。解析完毕后，`client` 会根据请求命令从命令表中获

取到对应的 redisCommand，如果对应 cmd 存在。则开始校验请求的参数，以及当前 server 的内存、磁盘及其他状态，完成校验后，然后真正开始执行 redisCommand 的处理函数，进行具体命令的执行，最后将执行结果作为响应写入 client 的写缓冲中。

命令回复处理器 sendReplyToClient

当 redis需要发送响应给client时，Redis 事件循环中会对client的连接socket注册写事件，这个写事件的处理函数就是sendReplyToClient。通过注册写事件，将 client 的socket与 AE\_WRITABLE 进行间接关联。当 Client fd 可进行写操作时，就会触发写事件，该函数就会将写缓冲中的数据发送给调用方。

Redis 中的时间事件是指需要在特定时间执行的事件。多个 Redis 中的时间事件构成 aeEventLoop 中的一个链表，供 Redis 在 ae 事件循环中轮询执行。

Redis 当前的主要时间事件处理函数有 2 个：

```
serverCron  
  
moduleTimerHandler
```

Redis 中的时间事件分为 2 类：

单次时间，即执行完毕后，该时间事件就结束了。

周期性事件，在事件执行完毕后，会继续设置下一次执行的事件，从而在时间到达后继续执行，并不断重复。

时间事件主要有 5 个属性组成。

事件 ID：Redis 为时间事件创建全局唯一 ID，该 ID 按从小到大的顺序进行递增。

执行时间 when\_sec 和 when\_ms：精确到毫秒，记录该事件的到达可执行时间。

时间事件处理器 timeProc：在时间事件到达时，Redis 会调用相应的 timeProc 处理事件。

关联数据 clientData：在调用 timeProc 时，需要使用该关联数据作为参数。

链表指针 prev 和 next：它用来将时间事件维护为双向链表，便于插入及查找所要执行的时间事件。

时间事件的处理是在事件循环中的 aeProcessEvents 中进行。执行过程是：

首先遍历所有的时间事件。

比较事件的时间和当前时间，找出可执行的时间事件。

然后执行时间事件的 `timeProc` 函数。

执行完毕后，对于周期性时间，设置时间新的执行时间；对于单次性时间，设置事件的 `ID` 为 `-1`，后续在事件循环中，下一次执行 `aeProcessEvents` 的时候从链表中删除。

## 第21讲：Redis读取请求数据后，如何进行协议解析和处理？

你好，我是你的缓存课老师陈波，欢迎进入第 21 课时“Redis 协议解析及处理”的学习。上一课时，我们学习了 Redis 事件驱动模型，接下来，看一下 Redis 是如何进行协议解析及处理的。

Redis 协议解析及处理

协议解析

上一课时讲到，请求命令进入，触发 IO 读事件后。client 会从连接文件描述符读取请求，并存入 client 的 query buffer 中。client 的读缓冲默认是 16KB，读取命令时，如果发现请求超过 1GB，则直接报异常，关闭连接。

client 读取完请求命令后，则根据 query buff 进行协议解析。协议解析时，首先查看协议的首字符。如果是 `*`，则解析为字符块数组类型，即 `MULTIBULK`。否则请求解析为 `INLINE` 类型。

`INLINE` 类型是以 `CRLF` 结尾的单行字符串，协议命令及参数以空格分隔。解析过程参考之前课程里分析的对应协议格式。协议解析完毕后，将请求参数个数存入 client 的 `argc` 中，将请求的具体参数存入 client 的 `argv` 中。

协议执行

请求命令解析完毕，则进入到协议执行部分。协议执行中，对于 `quit` 指令，直接返回 `OK`，设置 `flag` 为回复后关闭连接。

对于非 `quit` 指令，以 client 中 `argv[0]` 作为命令，从 server 中的命令表中找到对应的 `redisCommand`。如果没有找到 `redisCommand`，则返回未知 `cmd` 异常。如果找到 `cmd`，则开始执行 `redisCommand` 中的 `proc` 函数，进行具体命令的执行。在命令执行完毕后，将响应写入 client 的写缓冲。并按配置和部署，将写指令分发给 `aof` 和 `slaves`。同时更新相关的统计数值。

## 第22讲：怎么认识和应用Redis内部数据结构？

上一课时，我们学习了 Redis 协议解析及处理，接下来，看一下 Redis 的内部数据结构是什么样的？

Redis 内部数据结构

RdeisDb

Redis 中所有数据都保存在 DB 中，一个 Redis 默认最多支持 16 个 DB。Redis 中的每个 DB 都对应一个 redisDb 结构，即每个 Redis 实例，默认有 16 个 redisDb。用户访问时，默认使用的是 0 号 DB，可以通过 select \$dbID 在不同 DB 之间切换。

redisDb 主要包括 2 个核心 dict 字典、3 个非核心 dict 字典、dbID 和其他辅助属性。2 个核心 dict 包括一个 dict 主字典和一个 expires 过期字典。主 dict 字典用来存储当前 DB 中的所有数据，它将 key 和各种数据类型的 value 关联起来，该 dict 也称 key space。过期字典用来存储过期时间 key，存的是 key 与过期时间的映射。日常的数据存储和访问基本都会访问到 redisDb 中的这两个 dict。

3 个非核心 dict 包括一个字段名叫 blocking\_keys 的阻塞 dict，一个字段名叫 ready\_keys 的解除阻塞 dict，还有一个是字段名叫 watched\_keys 的 watch 监控 dict。

在执行 Redis 中 list 的阻塞命令 blpop、brpop 或者 brpoplpush 时，如果对应的 list 列表为空，Redis 就会将对应的 client 设为阻塞状态，同时将该 client 添加到 DB 中 blocking\_keys 这个阻塞 dict。所以该 dict 存储的是处于阻塞状态的 key 及 client 列表。

当有其他调用方在向某个 key 对应的 list 中增加元素时，Redis 会检测是否有 client 阻塞在这个 key 上，即检查 blocking\_keys 中是否包含这个 key，如果有则会将这个 key 加入 read\_keys 这个 dict 中。同时也会将这个 key 保存到 server 中的一个名叫 read\_keys 的列表中。这样可以高效、不重复的插入及轮询。

当 client 使用 watch 指令来监控 key 时，这个 key 和 client 就会被保存到 watched\_keys 这个 dict 中。redisDb 中可以保存所有的数据类型，而 Redis 中所有数据类型都是存放在一个叫 redisObject 的结构中。

redisObject

redisObject 由 5 个字段组成。

type: 即 Redis 对象的数据类型，目前支持 7 种 type 类型，分别为

OBJ\_STRING

OBJ\_LIST

OBJ\_SET

OBJ\_ZSET

OBJ\_HASH

OBJ\_MODULE

OBJ\_STREAM

encoding: Redis 对象的内部编码方式，即内部数据结构类型，目前支持 10 种编码方式包括

OBJ\_ENCODING\_RAW

OBJ\_ENCODING\_INT

OBJ\_ENCODING\_HT

OBJ\_ENCODING\_ZIPLIST 等。

LRU: 存储的是淘汰数据用的 LRU 时间或 LFU 频率及时间的数据。

refcount: 记录 Redis 对象的引用计数，用来表示对象被共享的次数，共享使用时加 1，不再使用时减 1，当计数为 0 时表明该对象没有被使用，就会被释放，回收内存。

ptr: 它指向对象的内部数据结构。比如一个代表 string 的对象，它的 ptr 可能指向一个 sds 或者一个 long 型整数。

## dict

前面讲到，Redis 中的数据实际是存在 DB 中的 2 个核心 dict 字典中的。实际上 dict 也是 Redis 的一种使用广泛的内部数据结构。

Redis 中的 dict，类似于 Memcached 中 hashtable。都可以用于 key 或元素的快速插入、更新和定位。dict 字典中，有一个长度为 2 的哈希表数组，日常访问用 0 号哈希表，如果 0 号哈希表元素过多，则分配一个 2 倍 0 号哈希表大小的空间给 1 号哈希表，然后进行逐步迁移，rehashidx 这个字段就是专门用来做标志迁移位置的。在哈希表操作中，采用单向链表来解决 hash 冲突问题。dict 中还有一个重要字段是 type，它用于保存 hash 函数及 key/value 赋值、比较函数。

dictht 中的 table 是一个 hash 表数组，每个桶指向一个 dictEntry 结构。dictht 采用 dictEntry 的单向链表来解决 hash 冲突问题。

dictht 是以 dictEntry 来存 key-value 映射的。其中 key 是 sds 字符串，value 为存储各种数据类型的 redisObject 结构。

dict 可以被 redisDb 用来存储数据 key-value 及命令操作的辅助信息。还可以用来作为一些 Redis 数据类型的内部数据结构。dict 可以作为 set 集合的内部数据结构。在哈希的元素数超过 512 个，或者哈希中 value 大于 64 字节，dict 还被用作为哈希类型的内部数据结构。

## sds

字符串是 Redis 中最常见的数据类型，其底层实现是简单动态字符串即 sds。简单动态字符串本质是一个 `char*`，内部通过 `sdshdr` 进行管理。`sdshdr` 有 4 个字段。`len` 为字符串实际长度，`alloc` 当前字节数组总共分配的内存大小。`flags` 记录当前字节数组的属性；`buf` 是存储字符串真正的值及末尾一个 `\0`。

sds 的存储 `buf` 可以动态扩展或收缩，字符串长度不用遍历，可直接获得，修改和访问都很方便。由于 sds 中字符串存在 `buf` 数组中，长度由 `len` 定义，而不像传统字符串遇 0 停止，所以 sds 是二进制安全的，可以存放任何二进制的数据。

简单动态字符串 sds 的获取字符串长度很方便，通过 `len` 可以直接得到，而传统字符串需要对字符串进行遍历，时间复杂度为  $O(n)$ 。

sds 相比传统字符串多了一个 `sdshdr`，对于大量很短的字符串，这个 `sdshdr` 还是一个不小的开销。在 3.2 版本后，sds 会根据字符串实际的长度，选择不同的数据结构，以更好的提升内存效率。当前 `sdshdr` 结构分为 5 种子类型，分别为 `sdshdr5`、`sdshdr8`、`sdshdr16`、`sdshdr32`、`sdshdr64`。其中 `sdshdr5` 只有 `flags` 和 `buf` 字段，其他几种类型的 `len` 和 `alloc` 采用从 `uint8_t` 到 `uint64_t` 的不同类型，以节省内存空间。

sds 可以作为字符串的内部数据结构，同时 sds 也是 `hyperloglog`、`bitmap` 类型的内部数据结构。  
`ziplist`

为了节约内存，并减少内存碎片，Redis 设计了 `ziplist` 压缩列表内部数据结构。压缩列表是一块连续的内存空间，可以连续存储多个元素，没有冗余空间，是一种连续内存数据块组成的顺序型内存结构。

`ziplist` 的结构如图所示，主要包括 5 个部分。

`zlbytes` 是压缩列表所占用的总内存字节数。

`zltail` 尾节点到起始位置的字节数。

`zllen` 总共包含的节点/内存块数。

`Entry` 是 `ziplist` 保存的各个数据节点，这些数据点长度随意。

`zlend` 是一个魔数 255，用来标记压缩列表的结束。

如图所示，一个包含 4 个元素的 `ziplist`，总占用字节是 100bytes，该 `ziplist` 的起始元素的指针是 `p`，`zltail` 是 80，则第 4 个元素的指针是 `P+80`。

压缩列表 `ziplist` 的存储节点 `entry` 的结构如图，主要有 6 个字段。

```
prevRawLen 是前置节点的长度；

preRawLenSize 编码 preRawLen 需要的字节数；

len 当前节点的长度；

lensize 编码 len 所需要的字节数；

encoding 当前节点所用的编码类型；

entryData 当前节点数据。
```

由于 ziplist 是连续紧凑存储，没有冗余空间，所以插入新的元素需要 realloc 扩展内存，所以如果 ziplist 占用空间太大，realloc 重新分配内存和拷贝的开销就会很大，所以 ziplist 不适合存储过多元素，也不适合存储过大的字符串。

因此只有在元素数和 value 数都不大的时候，ziplist 才作为 hash 和 zset 的内部数据结构。其中 hash 使用 ziplist 作为内部数据结构的限制时，元素数默认不超过 512 个，value 值默认不超过 64 字节。可以通过修改配置来调整 hash\_max\_ziplist\_entries、hash\_max\_ziplist\_value 这两个阈值的大小。

zset 有序集合，使用 ziplist 作为内部数据结构的限制元素数默认不超过 128 个，value 值默认不超过 64 字节。可以通过修改配置来调整 zset\_max\_ziplist\_entries 和 zset\_max\_ziplist\_value 这两个阈值的大小。

quicklist

Redis 在 3.2 版本之后引入 quicklist，用以替换 linkedlist。因为 linkedlist 每个节点有前后指针，要占用 16 字节，而且每个节点独立分配内存，很容易加剧内存的碎片化。而 ziplist 由于紧凑型存储，增加元素需要 realloc，删除元素需要内存拷贝，天然不适合元素太多、value 太大的存储。

而 quicklist 快速列表应运而生，它是一个基于 ziplist 的双向链表。将数据分段存储到 ziplist，然后将这些 ziplist 用双向指针连接。快速列表的结构如图所示。

```
head、tail 是两个指向第一个和最后一个 ziplist 节点的指针。

count 是 quicklist 中所有的元素个数。

len 是 ziplist 节点的个数。

compress 是 LZF 算法的压缩深度。
```

快速列表中，管理 ziplist 的是 quicklistNode 结构。quicklistNode 主要包含一个 prev/next 双向指针，以及一个 ziplist 节点。单个 ziplist 节点可以存放多个元素。



快速列表从头尾读写数据很快，时间复杂度为  $O(1)$ 。也支持从中间任意位置插入或读写元素，但速度较慢，时间复杂度为  $O(n)$ 。快速列表当前主要作为 list 列表的内部数据结构。

zskiplist

跳跃表 zskiplist 是一种有序数据结构，它通过在每个节点维持多个指向其他节点的指针，从而可以加速访问。跳跃表支持平均  $O(\log N)$  和最差  $O(n)$  复杂度的节点查找。在大部分场景，跳跃表的效率和平衡树接近，但跳跃表的实现比平衡树要简单，所以不少程序都用跳跃表来替换平衡树。

如果 sorted set 类型的元素数比较多或者元素比较大，Redis 就会选择跳跃表来作为 sorted set 有序集合的内部数据结构。

跳跃表主要由 zskipList 和节点 zskiplistNode 构成。zskiplist 结构如图，header 指向跳跃表的表头节点。tail 指向跳跃表的表尾节点。length 表示跳跃表的长度，它是跳跃表中不包含表头节点的节点数量。level 是目前跳跃表内，除表头节点外的所有节点中，层数最大的那个节点的层数。

跳跃表的节点 zskiplistNode 的结构如图所示。ele 是节点对应的 sds 值，在 zset 有序集合中就是集合中的 field 元素。score 是节点的分数，通过 score，跳跃表中的节点从小到大依次排列。backward 是指向当前节点的前一个节点的指针。level 是节点中的层，每个节点一般有多个层。每个 level 层都带有两个属性，一个是 forward 前进指针，它用于指向表尾方向的节点；另外一个 span 跨度，它是指 forward 指向的节点到当前节点的距离。

如图所示是一个跳跃表，它有 3 个节点。对应的元素值分别是 S1、S2 和 S3，分数值依次为 1.0、3.0 和 5.0。其中 S3 节点的 level 最大是 5，跳跃表的 level 是 5。header 指向表头节点，tail 指向表尾节点。在查到元素时，累加路径上的跨度即得到元素位置。在跳跃表中，元素必须是唯一的，但 score 可以相同。相同 score 的不同元素，按照字典序进行排序。

在 sorted set 数据类型中，如果元素数较多或元素长度较大，则使用跳跃表作为内部数据结构。默认元素数超过 128 或者最大元素的长度超过 64，此时有序集合就采用 zskiplist 进行存储。由于 geo 也采用有序集合类型来存储地理位置名称和位置 hash 值，所以在超过相同阈值后，也采用跳跃表进行存储。

Redis 主要的内部数据结构讲完了，接下来整体看一下，之前讲的 8 种数据类型，具体都是采用哪种内部数据结构来存储的。

首先，对于 string 字符串，Redis 主要采用 sds 来进行存储。而对于 list 列表，Redis 采用 quicklist 进行存储。对于 set 集合类型，Redis 采用 dict 来进行存储。对于 sorted set 有序集合类型，如果元素数小于 128 且元素长度小于 64，则使用 ziplist 存储，否则使用 zskiplist 存储。对于哈希类型，如果元素数小于 512，并且元素长度小于 64，则用 ziplist 存储，否则使用 dict 字典存储。对于 hyperloglog，采用 sds 简单动态字符串存储。对于 geo，如果位置数小于 128，则使用 ziplist 存储，否则使用 zskiplist 存储。最后对于 bitmap，采用 sds 简单动态字符串存储。

除了这些主要的内部数据结构，还有在特殊场景下也会采用一些其他内部结构存储，比如，如果操作的字符串都是整数，同时指令是 incr、decr 等，会对字符串采用 long 型整数存储，这些场景比较特殊，限于时间关系，这里不做进一步阐述。

## 第八章：Redis进阶（下）

### 第23讲：Redis是如何淘汰key的？

你好，我是你的缓存课老师陈波，欢迎进入第 23 课时“Redis 淘汰策略”的学习。本课时我们主要学习 Redis 淘汰原理、淘汰方式、以及 8 种淘汰策略等内容。

#### 淘汰原理

首先我们来学习 Redis 的淘汰原理。

系统线上运行中，内存总是昂贵且有限的，在数据总量远大于 Redis 可用的内存总量时，为了最大限度的提升访问性能，Redis 中只能存放最新最热的有效数据。

当 key 过期后，或者 Redis 实际占用的内存超过阈值后，Redis 就会对 key 进行淘汰，删除过期的或者不活跃的 key，回收其内存，供新的 key 使用。Redis 的内存阈值是通过 maxmemory 设置的，而超过内存阈值后的淘汰策略，是通过 maxmemory-policy 设置的，具体的淘汰策略后面会进行详细介绍。Redis 会在 2 种场景下对 key 进行淘汰，第一种是在定期执行 serverCron 时，检查淘汰 key；第二种是在执行命令时，检查淘汰 key。

第一种场景，Redis 定期执行 serverCron 时，会对 DB 进行检测，清理过期 key。清理流程如下。首先轮询每个 DB，检查其 expire dict，即带过期时间的过期 key 字典，从所有带过期时间的 key 中，随机选取 20 个样本 key，检查这些 key 是否过期，如果过期则清理删除。如果 20 个样本中，超过 5 个 key 都过期，即过期比例大于 25%，就继续从该 DB 的 expire dict 过期字典中，再随机取样 20 个 key 进行过期清理，持续循环，直到选择的 20 个样本 key 中，过期的 key 数小于等于 5，当前这个 DB 则清理完毕，然后继续轮询下一个 DB。

在执行 serverCron 时，如果在某个 DB 中，过期 dict 的填充率低于 1%，则放弃对该 DB 的取样检查，因为效率太低。如果 DB 的过期 dict 中，过期 key 太多，一直持续循环回收，会占用大量主线程时间，所以 Redis 还设置了一个过期时间。这个过期时间根据 serverCron 的执行频率来计算，5.0 版本及之前采用慢循环过期策略，默认是 25ms，如果回收超过 25ms 则停止，6.0 非稳定版本采用快循环策略，过期时间为 1ms。

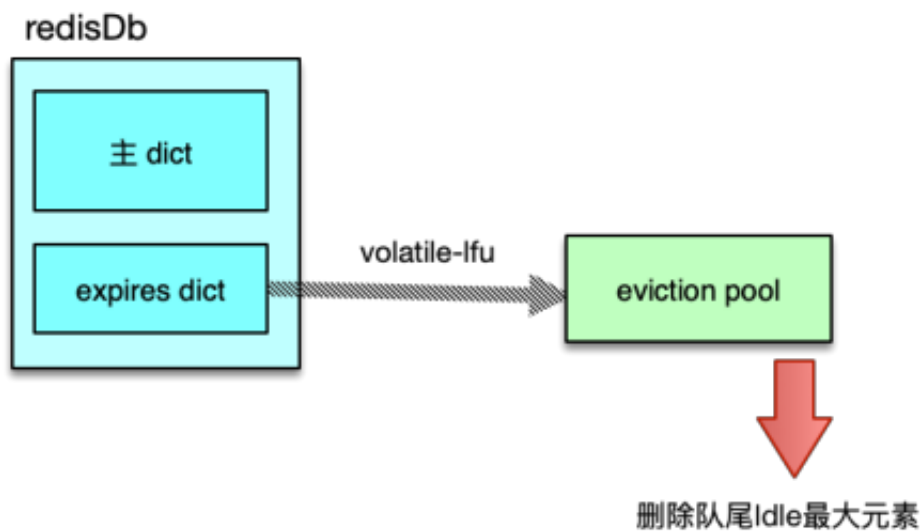
第二种场景，Redis 在执行命令请求时。会检查当前内存占用是否超过 maxmemory 的数值，如果超过，则按照设置的淘汰策略，进行删除淘汰 key 操作。

#### 淘汰方式

Redis 中 key 的淘汰方式有两种，分别是同步删除淘汰和异步删除淘汰。在 serverCron 定期清理过期 key 时，如果设置了延迟过期配置 lazyfree-lazy-expire，会检查 key 对应的 value 是否为多元素的复合类型，即是否是 list 列表、set 集合、zset 有序集合和 hash 中的一种，并且 value 的元素数大于 64，则在将 key 从 DB 中 expire dict 过期字典和主 dict 中删除后，value 存放到 BIO 任务队列，由 BIO 延迟删除线程异步回收；否则，直接从 DB 的 expire dict 和主 dict 中删除，并回收 key、value 所占用的空间。在执行命令时，如果设置了 lazyfree-lazy-eviction，在淘汰 key 时，也采用前面类似的检测方法，对于元素数大于 64 的 4 种复合类型，使用 BIO 线程异步删除，否则采用同步直接删除。

#### 淘汰策略

Redis 提供了 8 种淘汰策略对 key 进行管理，而且还引入基于样本的 eviction pool，来提升剔除的准确性，确保在保持最大性能的前提下，剔除最不活跃的 key。eviction pool 主要对 LRU、LFU，以及过期 dict ttl 内存管理策略生效。处理流程为，当 Redis 内存占用超过阈值后，按策略从主 dict 或者带过期时间的 expire dict 中随机选择 N 个 key，N 默认是 5，计算每个 key 的 idle 值，按 idle 值从小到大的顺序插入 evictionPool 中，然后选择 idle 最大的那个 key，进行淘汰。



选择淘汰策略时，可以通过配置 Redis 的 maxmemory 设置最大内存，并通 maxmemory\_policy 设置超过最大内存后的处理策略。如果 maxmemory 设为 0，则表明对内存使用没有任何限制，可以持续存放数据，适合作为存储，来存放数据量较小的业务。如果数据量较大，就需要估算热数据容量，设置一个适当的值，将 Redis 作为一个缓存而非存储来使用。

Redis 提供了 8 种 maxmemory\_policy 淘汰策略来应对内存超过阈值的情况。

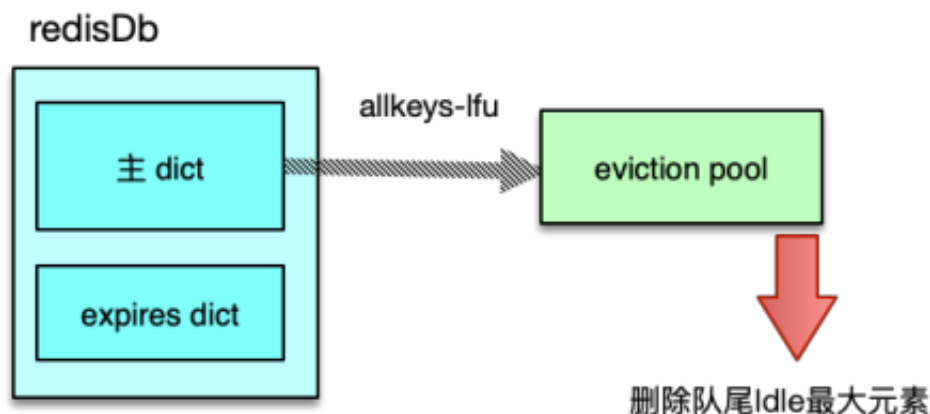
第一种淘汰策略是 noeviction，它是 Redis 的默认策略。在内存超过阈值后，Redis 不做任何清理工作，然后对所有写操作返回错误，但对读请求正常处理。noeviction 适合数据量不大的业务场景，将关键数据存入 Redis 中，将 Redis 当作 DB 来使用。

第二种淘汰策略是 volatile-lru，它对带过期时间的 key 采用最近最少访问算法来淘汰。使用这种策略，Redis 会从 redisDb 的 expire dict 过期字典中，首先随机选择 N 个 key，计算 key 的空闲时间，然后插入 evictionPool 中，最后选择空闲时间最久的 key 进行淘汰。这种策略适合的业务场景是，需要淘汰的 key 带有过期时间，且有冷热区分，从而可以淘汰最久没有访问的 key。

第三种策略是 volatile-lfu，它对带过期时间的 key 采用最近最不经常使用的算法来淘汰。使用这种策略时，Redis 会从 redisDb 中的 expire dict 过期字典中，首先随机选择 N 个 key，然后根据其 value 的 lru 值，计算 key 在一段时间内的使用频率相对值。对于 lfu，要选择使用频率最小的 key，为了沿用 evictionPool 的 idle 概念，Redis 在计算 lfu 的 Idle 时，采用 255 减去使用频率相对值，从而确保 Idle 最大的 key 是使用次数最小的 key，计算 N 个 key 的 Idle 值后，插入 evictionPool，最后选择 Idle 最大，即使用频率最小的 key，进行淘汰。这种策略也适合大多数 key 带过期时间且有冷热区分的业务场景。

第四种策略是 volatile-ttl，它是对带过期时间的 key 中选择最早要过期的 key 进行淘汰。使用这种策略时，Redis 也会从 redisDb 的 expire dict 过期字典中，首先随机选择 N 个 key，然后用最大无符号 long 值减去 key 的过期时间来作为 Idle 值，计算 N 个 key 的 Idle 值后，插入 evictionPool，最后选择 Idle 最大，即最快就要过期的 key，进行淘汰。这种策略适合，需要淘汰的 key 带过期时间，且有按时间

冷热区分的业务场景。



第五种策略是 volatile-random，它是对带过期时间的 key 中随机选择 key 进行淘汰。使用这种策略时，Redis 从 redisDb 的 expire dict 过期字典中，随机选择一个 key，然后进行淘汰。如果需要淘汰的 key 有过期时间，没有明显热点，主要被随机访问，那就适合选择这种淘汰策略。

第六种策略是 allkey-lru，它是对所有 key，而非仅仅带过期时间的 key，采用最近最久没有使用的算法来淘汰。这种策略与 volatile-lru 类似，都是从随机选择的 key 中，选择最长时间没有被访问的 key 进行淘汰。区别在于，volatile-lru 是从 redisDb 中的 expire dict 过期字典中选择 key，而 allkey-lru 是从所有的 key 中选择 key。这种策略适合，需要对所有 key 进行淘汰，且数据有冷热读写区分的业务场景。

第七种策略是 allkeys-lfu，它也是针对所有 key 采用最近最经常使用的算法来淘汰。这种策略与 volatile-lfu 类似，都是在随机选择的 key 中，选择访问频率最小的 key 进行淘汰。区别在于，volatile-lfu 从 expire dict 过期字典中选择 key，而 allkeys-lfu 是从主 dict 中选择 key。这种策略适合的场景是，需要从所有的 key 中进行淘汰，但数据有冷热区分，且越热的数据访问频率越高。

最后一种策略是 allkeys-random，它是针对所有 key 进行随机算法进行淘汰。它也是从主 dict 中随机选择 key，然后进行删除回收。如果要从所有的 key 中进行淘汰，并且 key 的访问没有明显热点，被随机访问，即可采用这种策略。

## 第24讲：Redis崩溃后，如何进行数据恢复的？

你好，我是你的缓存课老师陈波，欢迎来到第 24 课时“Redis 崩溃后，如何进行数据恢复”的学习。本课时我们主要学习通过 RDB、AOF、混合存储等数据持久化方案来解决如何进行数据恢复的问题。

Redis 持久化是一个将内存数据转储到磁盘的过程。Redis 目前支持 RDB、AOF，以及混合存储三种模式。

RDB

Redis 的 RDB 持久化是以快照的方式将内存数据存储到磁盘。在需要进行 RDB 持久化时，Redis 会将内存中的所有数据以二进制的格式落地，每条数据存储的内容包括过期时间、数据类型、key，以及 value。当 Redis 重启时，如果 appendonly 关闭，则会读取 RDB 持久化生成的二进制文件进行数据恢复。

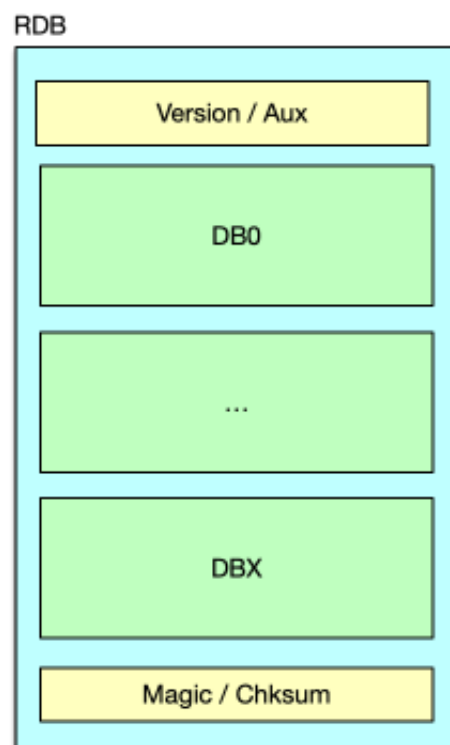
触发构建 RDB 的场景主要有以下四种。

第一种场景是通过 `save` 或 `bgsave` 命令进行主动 RDB 快照构建。它是由调用方调用 `save` 或 `bgsave` 指令进行触发的。

第二种场景是利用配置 `save m n` 来进行自动快照生成。它是指在 `m` 秒中，如果插入或变更 `n` 个 `key`，则自动触发 `bgsave`。这个配置可以设置多个配置行，以便组合使用。由于峰值期间，Redis 的压力大，变更的 `key` 也比较多，如果再进行构建 RDB 的操作，会进一步增加机器负担，对调用方请求会有一些影响，所以线上使用时需要谨慎。

第三种场景是主从复制，如果从库需要进行全量复制，此时主库也会进行 `bgsave` 生成一个 RDB 快照。

第四种场景是在运维执行 `flushall` 清空所有数据，或执行 `shutdown` 关闭服务时，也会触发 Redis 自动构建 RDB 快照。



`save` 是在主进程中进行 RDB 持久化的，持久化期间 Redis 处于阻塞状态，不处理任何客户请求，所以一般使用较少。而 `bgsave` 是 `fork` 一个子进程，然后在子进程中构建 RDB 快照，构建快照的过程不直接影响用户的访问，但仍然会增加机器负载。线上 Redis 快照备份，一般会选择凌晨低峰时段，通过 `bgsave` 主动触发进行备份。

RDB 快照文件主要由 3 部分组成。

第一部分是 RDB 头部，主要包括 RDB 的版本，以及 Redis 版本、创建日期、占用内存等辅助信息。

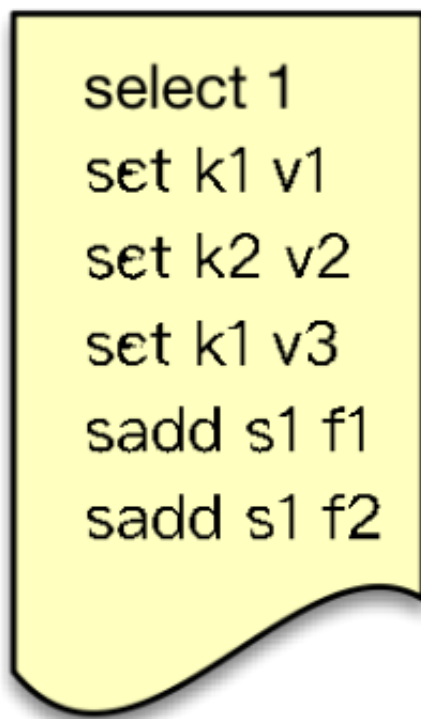
第二部分是各个 RedisDB 的数据。存储每个 RedisDB 时，会首先记录当前 RedisDB 的 `DBID`，然后记录主 `dict` 和 `expire dict` 的记录数量，最后再轮询存储每条数据记录。存储数据记录时，如果数据有过期时间，首先记录过期时间。如果 Redis 的 `maxmemory_policy` 过期策略采用 LRU 或者 LFU，还会将 `key` 对应的 LRU、LFU 值进行落地，最后记录数据的类型、`key`，以及 `value`。

第三部分是 RDB 的尾部。RDB 尾部，首先存储 Redis 中的 Lua 脚本等辅助信息。然后存储 EOF 标记，即值为 255 的字符。最后存 RDB 的 `cksum`。

至此，RDB 就落地完毕。

RDB 采用二进制方式存储内存数据，文件小，且启动时恢复速度快。但构建 RDB 时，一个快照文件只能存储，构建时刻的内存数据，无法记录之后的数据变更。构建 RDB 的过程，即便在子进程中进行，但仍然属于 CPU 密集型的操作，而且每次落地全量数据，耗时也比较长，不能随时进行，特别是不能在高峰期进行。由于 RDB 采用二进制存储，可读性差，而且由于格式固定，不同版本之间可能存在兼容性问题。

AOF



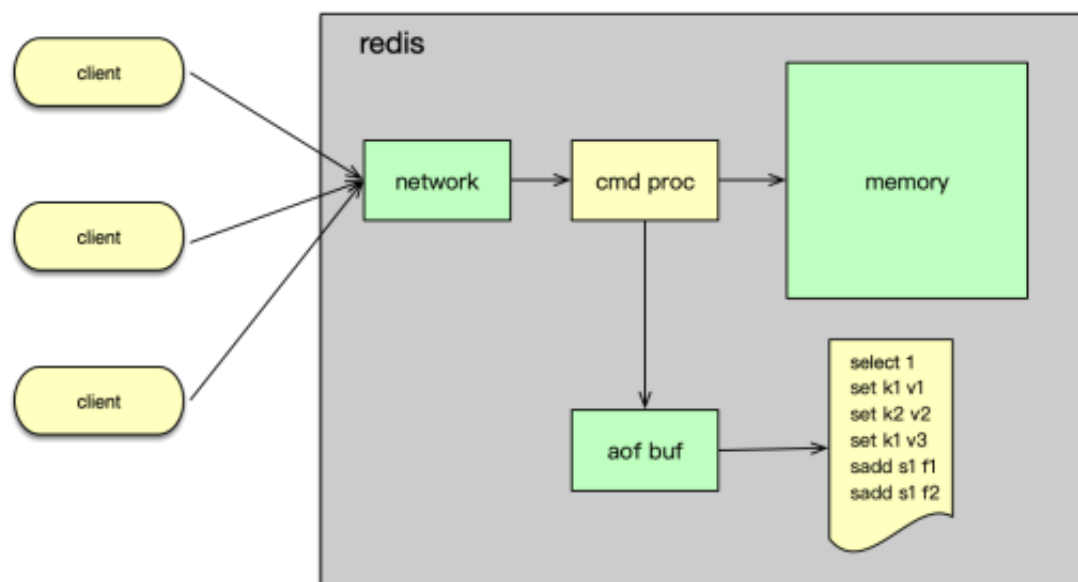
```
select 1
set k1 v1
set k2 v2
set k1 v3
sadd s1 f1
sadd s1 f2
```

Redis 的 AOF 持久化是以命令追加的方式进行数据落地的。通过打开 appendonly 配置，Redis 将每一个写指令追加到磁盘 AOF 文件，从而及时记录内存数据的最新状态。这样即便 Redis 被 crash 或异常关闭后，再次启动，也可以通过加载 AOF，来恢复最新的全量数据，基本不会丢失数据。

AOF 文件中存储的协议是写指令的 multibulk 格式，这是 Redis 的标准协议格式，所以不同的 Redis 版本均可解析并处理，兼容性很好。

但是，由于 Redis 会记录所有写指令操作到 AOF，大量的中间状态数据，甚至被删除的过期数据，都会存在 AOF 中，冗余度很大，而且每条指令还需通过加载和执行来进行数据恢复，耗时会比较大。

AOF 数据的落地流程如下。Redis 在处理完写指令后，首先将写指令写入 AOF 缓冲，然后通过 server\_cron 定期将 AOF 缓冲写入文件缓冲。最后按照配置策略进行 fsync，将文件缓冲的数据真正同步写入磁盘。



Redis 通过 `appendfsync` 来设置三种不同的同步文件缓冲策略。

第一种配置策略是 `no`，即 Redis 不主动使用 `fsync` 进行文件数据同步落地，而是由操作系统的 `write` 函数去确认同步时间，在 Linux 系统中大概每 30 秒会进行一次同步，如果 Redis 发生 `crash`，就会造成大量的数据丢失。

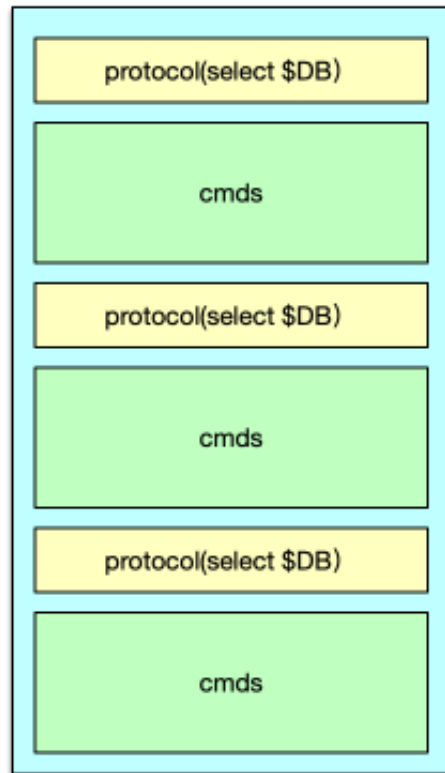
第二种配置策略是 `always`，即每次将 AOF 缓冲写入文件，都会调用 `fsync` 强制将内核数据写入文件，安全性最高，但性能上会比较低效，而且由于频繁的 IO 读写，磁盘的寿命会大大降低。

第三种配置策略是 `everysec`。即每秒通过 BIO 线程进行一次 `fsync`。这种策略在安全性、性能，以及磁盘寿命之间做较好的权衡，可以较好的满足线上业务需要。

随着时间的推移，AOF 持续记录所有的写指令，AOF 会越来越大，而且会充斥大量的中间数据、过期数据，为了减少无效数据，提升恢复时间，可以定期对 AOF 进行 `rewrite` 操作。

AOF 的 `rewrite` 操作可以通过运维执行 `bgrewriteaof` 命令来进行，也可以通过配置重写策略进行，由 Redis 自动触发进行。当对 AOF 进行 `rewrite` 时，首先会 `fork` 一个子进程。子进程轮询所有 RedisDB 快照，将所有内存数据转为 `cmd`，并写入临时文件。在子进程 `rewriteaof` 时，主进程可以继续执行用户请求，执行完毕后将写指令写入旧的 AOF 文件和 `rewrite` 缓冲。子进程将 RedisDB 中数据落地完毕后，通知主进程。主进程从而将 AOF `rewrite` 缓冲数据写入 AOF 临时文件，然后用新的 AOF 文件替换旧的 AOF 文件，最后通过 BIO 线程异步关闭旧的 AOF 文件。至此，AOF 的 `rewrite` 过程就全部完成了。





AOF 重写的过程，是一个轮询全部 RedisDB 快照，逐一落地的过程。每个 DB，首先通过 `select $db` 来记录待落的 DBID。然后通过命令记录每个 key/value。对于数据类型为 SDS 的 value，可以直接落地。但如果 value 是聚合类型，则会将所有元素设为批量添加指令，进行落地。

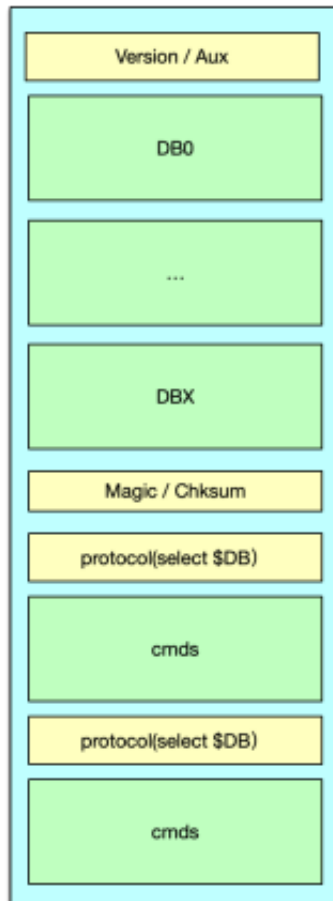
对于 list 列表类型，通过 `Rpush` 指令落地所有列表元素。对于 set 集合，会用 `Sadd` 落地所有集合元素。对于 Zset 有序集合，会用 `Zadd` 落地所有元素，而对于 Hash 会用 `Hmset` 落地所有哈希元素。如果数据带过期时间，还会通过 `pexpireat` 来记录数据的过期时间。

AOF 持久化的优势是可以记录全部的最新内存数据，最多也就是 1-2 秒的数据丢失。同时 AOF 通过 Redis 协议来追加记录数据，兼容性高，而且可以持续轻量级的保存最新数据。最后因为是直接通过 Redis 协议存储，可读性也比较好。

AOF 持久化的不足是随着时间的增加，冗余数据增多，文件会持续变大，而且数据恢复需要读取所有命令并执行，恢复速度相对较慢。

混合持久化





Redis 在 4.0 版本之后，引入了混合持久化方式，而且在 5.0 版本后默认开启。前面讲到 RDB 加载速度快，但构建慢，缺少最新数据。AOF 持续追加最新写记录，可以包含所有数据，但冗余大，加载速度慢。混合模式一体化使用 RDB 和 AOF，综合 RDB 和 AOF 的好处。即可包含全量数据，加载速度也比较快。可以使用 `aof-use-rdb-preamble` 配置来明确打开混合持久化模式。

混合持久化也是通过 `bgrewriteaof` 来实现的。当启用混合存储后，进行 `bgrewriteaof` 时，主进程首先依然是 `fork` 一个子进程，子进程首先将内存数据以 RDB 的二进制格式写入 AOF 临时文件中。然后，再将落地期间缓冲的新增写指令，以命令的方式追加到临时文件。然后再通知主进程落地完毕。主进程将临时文件修改为 AOF 文件，并关闭旧的 AOF 文件。这样主体数据以 RDB 格式存储，新增指令以命令方式追加的混合存储方式进行持久化。后续执行的任务，以正常的命令方式追加到新的 AOF 文件即可。

混合持久化综合了 RDB 和 AOF 的优缺点，优势是包含全量数据，加载速度快。不足是头部的 RDB 格式兼容性和可读性较差。

## 第25讲：Redis是如何处理容易超时的系统调用的？

本课时我们主要学习通过 BIO 线程解决处理容易超时的系统调用问题，以及 BIO 线程处理的任务与处理流程等内容。

BIO 线程简介

Redis 在运行过程中，不可避免的会产生一些运行慢的、容易引发阻塞的任务，如将内核中的文件缓冲同步到磁盘中、关闭文件，都会引发短时阻塞，还有一些大 key，如一些元素数高达万级或更多的聚合类元素，在删除时，由于所有元素需要逐一释放回收，整个过程耗时也会比较长。而 Redis 的核心处理线程是单进程单线程模型，所有命令的接受与处理、数据淘汰等都在主线程中进行，这些任务处理速度非常快。如果核心单线程还要处理那些慢任务，在处理期间，势必会阻塞用户的正常请求，导致服务卡顿。为此，Redis 引入了 BIO 后台线程，专门处理那些慢任务，从而保证和提升主线程的处理能力。

Redis 的 BIO 线程采用生产者-消费者模型。主线程是生产者，生产各种慢任务，然后存放到任务队列中。BIO 线程是消费者，从队列获取任务并进行处理。如果生产者生产任务过快，队列可用于缓冲这些任务，避免负荷过载或数据丢失。如果消费者处理速度很快，处理完毕后就可以安静的等待，不增加额外的性能开销。再次，有新任务时，主线程通过条件变量来通知 BIO 线程，这样 BIO 线程就可以再次执行任务。

### BIO 处理任务

Redis 启动时，会创建三个任务队列，并对应构建 3 个 BIO 线程，三个 BIO 线程与 3 个任务队列之间一一对应。BIO 线程分别处理如下 3 种任务。

`close` 关闭文件任务。`rewriteaof` 完成后，主线程需要关闭旧的 AOF 文件，就向 `close` 队列插入一个旧 AOF 文件的关闭任务。由 `close` 线程来处理。

`fsync` 任务。Redis 将 AOF 数据缓冲写入文件内核缓冲后，需要定期将系统内核缓冲数据写入磁盘，此时可以向 `fsync` 队列写入一个同步文件缓冲的任务，由 `fsync` 线程来处理。

`lazyfree` 任务。Redis 在需要淘汰元素数大于 64 的聚合类数据类型时，如列表、集合、哈希等，就往延迟清理队列中写入待回收的对象，由 `lazyfree` 线程后续进行异步回收。

### BIO 处理流程

BIO 线程的整个处理流程如图所示。当主线程有慢任务需要异步处理时。就会向对应的任务队列提交任务。提交任务时，首先申请内存空间，构建 BIO 任务。然后对队列锁进行加锁，在队列尾部追加新的 BIO 任务，最后尝试唤醒正在等待任务的 BIO 线程。

BIO 线程启动时或持续处理完所有任务，发现任务队列为空后，就会阻塞，并等待新任务的到来。当主线程有新任务后，主线程会提交任务，并唤醒 BIO 线程。BIO 线程随后开始轮询获取新任务，并进行处理。当处理完所有 BIO 任务后，则再次进入阻塞，等待下一轮唤醒。

## 第26讲：如何大幅成倍提升Redis处理性能？

本课时我们主要学习如何通过 Redis 多线程来大幅提升性能，涉及主线程与 IO 线程、命令处理流程，以及多线程方案的优劣等内容。

### 主线程

Redis 自问世以来，广受好评，应用广泛。但相比，Memcached 单实例压测 TPS 可以高达百万，线上可以稳定跑 20~40 万而言，Redis 的单实例压测 TPS 不过 10~12 万，线上一般最高也就 2~4 万，仍相差一个数量级。

Redis 慢的主要原因是单进程单线程模型。虽然一些重量级操作也进行了分拆，如 RDB 的构建在子进程中进行，文件关闭、文件缓冲同步，以及大 key 清理都放在 BIO 线程异步处理，但还远远不够。线上 Redis 处理用户请求时，十万级的 client 挂在一个 Redis 实例上，所有的事件处理、读请求、命令解析、命令执行，以及最后的响应回复，都由主线程完成，纵然是 Redis 各种极端优化，巧妇难为无米之炊，一个线程的处理能力始终是有上限的。当前服务器 CPU 大多是 16 核到 32 核以上，Redis 日常运行主要只使用 1 个核心，其他 CPU 核就没有被很好的利用起来，Redis 的处理性能也就无法有效地提升。而 Memcached 则可以按照服务器的 CPU 核心数，配置数十个线程，这些线程并发进行 IO 读写、任务处理，处理性能可以提高一个数量级以上。

## IO 线程

面对性能提升困境，虽然 Redis 作者不以为然，认为可以通过多部署几个 Redis 实例来达到类似多线程的效果。但多实例部署则带来了运维复杂的问题，而且单机多实例部署，会相互影响，进一步增大运维的复杂度。为此，社区一直有种声音，希望 Redis 能开发多线程版本。

因此，Redis 即将在 6.0 版本引入多线程模型，当前代码在 unstable 版本中，6.0 版本预计在明年发布。Redis 的多线程模型，分为主线程和 IO 线程。

因为处理命令请求的几个耗时点，分别是请求读取、协议解析、协议执行，以及响应回复等。所以 Redis 引入 IO 多线程，并发地进行请求命令的读取、解析，以及响应的回复。而其他的所有任务，如事件触发、命令执行、IO 任务分发，以及其他各种核心操作，仍然在主线程中进行，也就说这些任务仍然由单线程处理。这样可以在最大程度不改变原处理流程的情况下，引入多线程。

## 命令处理流程

Redis 6.0 的多线程处理流程如图所示。主线程负责监听端口，注册连接读事件。当有新连接进入时，主线程 accept 新连接，创建 client，并为新连接注册请求读事件。

当请求命令进入时，在主线程触发读事件，主线程此时并不进行网络 IO 的读取，而将该连接所在的 client 加入待读取队列中。Redis 的 Ae 事件模型在循环中，发现待读取队列不为空，则将所有待读取请求的 client 依次分派给 IO 线程，并自旋检查等待，等待 IO 线程读取所有的网络数据。所谓自旋检查等待，也就是指主线程持续死循环，并在循环中检查 IO 线程是否读完，不做其他任何任务。只有发现 IO 线程读完所有网络数据，才停止循环，继续后续的任务处理。

一般可以配置多个 IO 线程，比如配置 4~8 个，这些 IO 线程发现待读取队列中有任务时，则开始并发处理。每个 IO 线程从对应列表获取一个任务，从里面的 client 连接中读取请求数据，并进行命令解析。当 IO 线程完成所有的请求读取，并完成解析后，待读取任务数变为 0。主线程就停止循环检测，开始依次执行 IO 线程已经解析的所有命令，每执行完毕一个命令，就将响应写入 client 写缓冲，这些 client 就变为待回复 client，这些待回复 client 被加入待回复列表。然后主线程将这些待回复 client，轮询分配给多个 IO 线程。然后再次自旋检测等待。

然后 IO 线程再次开始并发执行，将不同 client 的响应缓冲写给 client。当所有响应全部处理完后，待回复的任务数变为 0，主线程结束自旋检测，继续处理后续的任务，以及新的读请求。

Redis 6.0 版本中新引入的多线程模型，主要是指可配置多个 IO 线程，这些线程专门负责请求读取、解析，以及响应的回复。通过 IO 多线程，Redis 的性能可以提升 1 倍以上。

## 多线程方案优劣

虽然多线程方案能提升1倍以上的性能，但整个方案仍然比较粗糙。首先所有命令的执行仍然在主线程中进行，存在性能瓶颈。然后所有的事件触发也是在主线程中进行，也依然无法有效使用多核心。而且，IO 读写为批处理读写，即所有 IO 线程先一起读完所有请求，待主线程解析处理完毕后，所有 IO 线程再一起回复所有响应，不同请求需要相互等待，效率不高。最后在 IO 批处理读写时，主线程自旋检测等待，效率更是低下，即便任务很少，也很容易把 CPU 打满。整个多线程方案比较粗糙，所以性能提升也很有限，也就 1~2 倍多一点而已。要想更大幅提升处理性能，命令的执行、事件的触发等都需要分拆到不同线程中进行，而且多线程处理模型也需要优化，各个线程自行进行 IO 读写和执行，互不干扰、等待与竞争，才能真正高效地利用服务器多核心，达到性能数量级的提升。

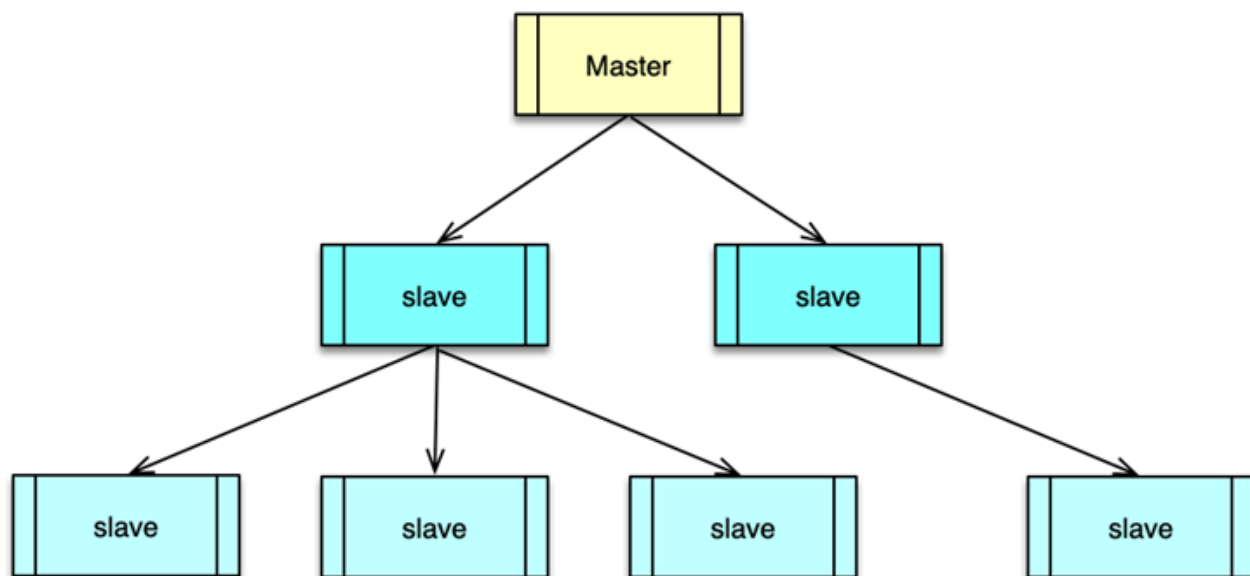
## 第九章：分布式Redis实战

### 第27讲：Redis是如何进行主从复制的？

本课时我们主要学习 Redis 复制原理，以及复制分析等内容。

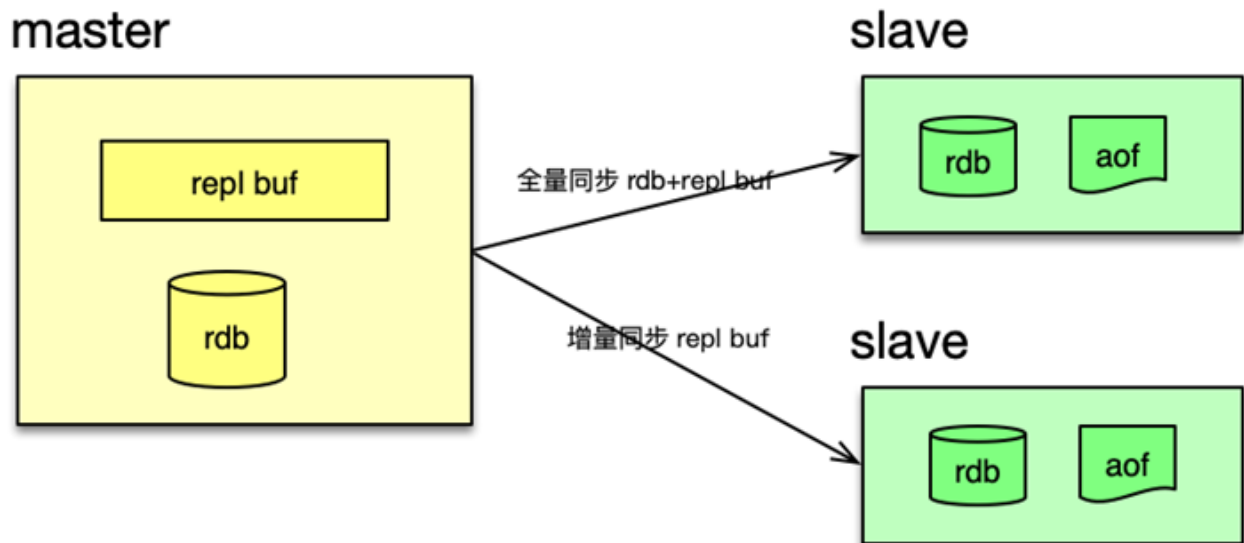
#### Redis 复制原理

为了避免单点故障，数据存储需要进行多副本构建。同时由于 Redis 的核心操作是单线程模型的，单个 Redis 实例能处理的请求 TPS 有限。因此 Redis 自面世起，基本就提供了复制功能，而且对复制策略不断进行优化。



通过数据复制，Redis 的一个 master 可以挂载多个 slave，而 slave 下还可以挂载多个 slave，形成多层嵌套结构。所有写操作都在 master 实例中进行，master 执行完毕后，将写指令分发给挂在自己下面的 slave 节点。slave 节点下如果有嵌套的 slave，会将收到的写指令进一步分发给挂在自己下面的 slave。通过多个 slave，Redis 的节点数据就可以实现多副本保存，任何一个节点异常都不会导致数据丢失，同时多 slave 可以 N 倍提升读性能。master 只写不读，这样整个 master-slave 组合，读写能力都可以得到大幅提升。

master 在分发写请求时，同时会将写指令复制一份存入复制积压缓冲，这样当 slave 短时间断开重连时，只要 slave 的复制位置点仍然在复制积压缓冲，则可以从之前的复制位置点之后继续进行复制，提升复制效率。



主库 master 和从库 slave 之间通过复制 id 进行匹配，避免 slave 挂到错误的 master。Redis 的复制分为全量同步和增量同步。Redis 在进行全量同步时，master 会将内存数据通过 bgsave 落地到 rdb，同时，将构建内存快照期间的写指令，存放到复制缓冲中，当 rdb 快照构建完毕后，master 将 rdb 和复制缓冲队列中的数据全部发送给 slave，slave 完全重新创建一份数据。这个过程，对 master 的性能损耗较大，slave 构建数据的时间也比较长，而且传递 rdb 时还会占用大量带宽，对整个系统的性能和资源的访问影响都比较大。而增量复制，master 只发送 slave 上次复制位置之后的写指令，不用构建 rdb，而且传输内容非常有限，对 master、slave 的负荷影响很小，对带宽的影响可以忽略，整个系统受影响非常小。

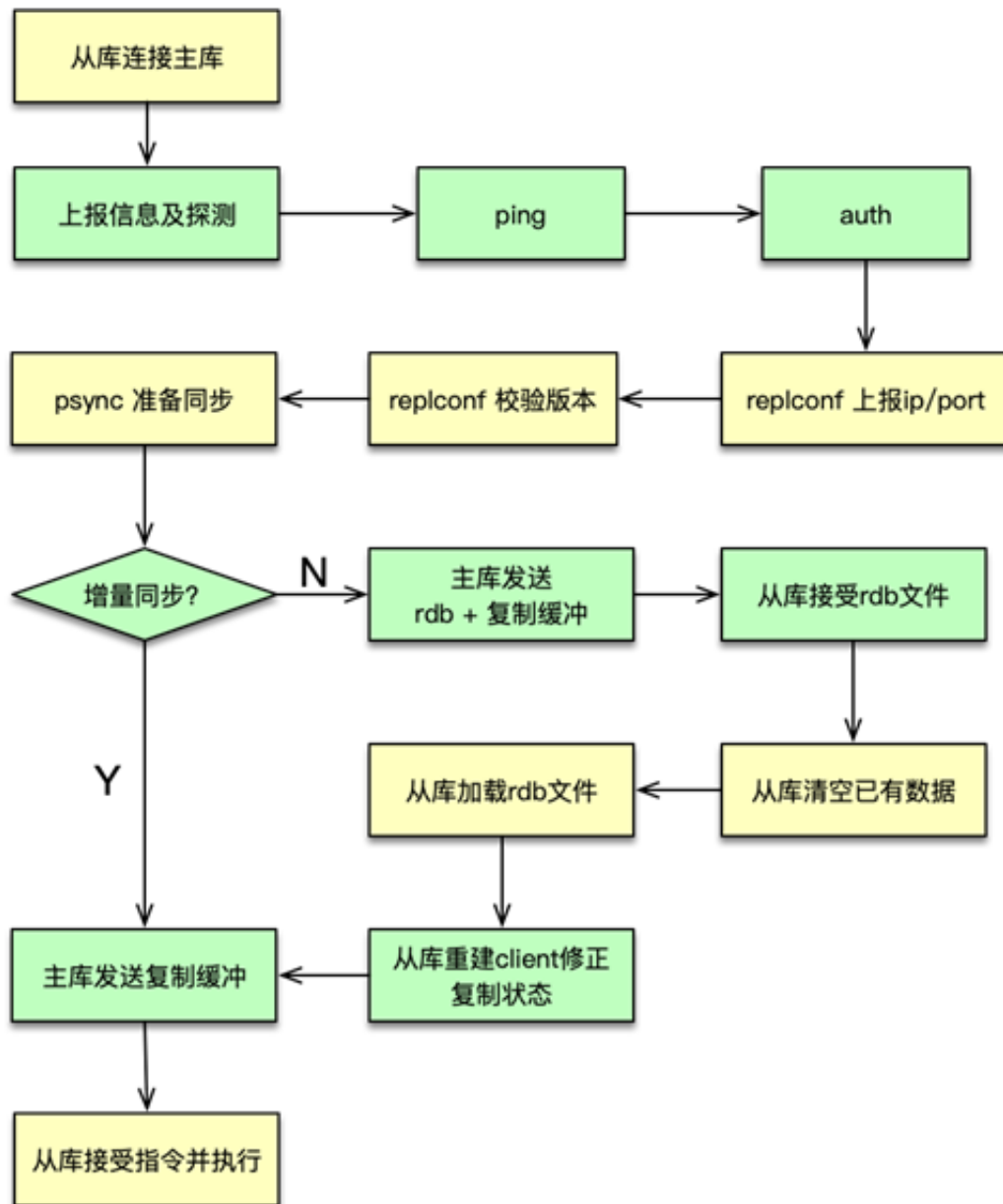
在 Redis 2.8 之前，Redis 基本只支持全量复制。在 slave 与 master 断开连接，或 slave 重启后，都需要进行全量复制。在 2.8 版本之后，Redis 引入 psync，增加了一个复制积压缓冲，在将写指令同步给 slave 时，会同时在复制积压缓冲中也写一份。在 slave 短时断开重连后，上报 master runid 及复制偏移量。如果 runid 与 master 一致，且偏移量仍然在 master 的复制缓冲积压中，则 master 进行增量同步。

但如果 slave 重启后，master runid 会丢失，或者切换 master 后，runid 会变化，仍然需要全量同步。因此 Redis 自 4.0 强化了 psync，引入了 psync2。在 psync2 中，主从复制不再使用 runid，而使用 replid（即复制 id）来作为复制判断依据。同时 Redis 实例在构建 rdb 时，会将 replid 作为 aux 辅助信息存入 rdb。重启时，加载 rdb 时即可得到 master 的复制 id。从而在 slave 重启后仍然可以增量同步。

在 psync2 中，Redis 每个实例除了会有一个复制 id 即 replid 外，还有一个 replid2。Redis 启动后，会创建一个长度为 40 的随机字符串，作为 replid 的初值，在建立主从连接后，会用 master 的 replid 替换自己的 replid。同时会用 replid2 存储上次 master 主库的 replid。这样切主时，即便 slave 汇报的复制 id 与新 master 的 replid 不同，但和新 master 的 replid2 相同，同时复制偏移仍然在复制积压缓冲区内，仍然可以实现增量复制。

Redis 复制分析

在设置 master、slave 时，首先通过配置或者命令 `slaveof no one` 将节点设置为主库。然后其他各个从库节点，通过 `slaveof $master_ip $master_port`，将其他从库挂在到 master 上。同样方法，还可以将 slave 节点挂载到已有的 slave 节点上。在准备开始数据复制时，slave 首先会主动与 master 创建连接，并上报信息。具体流程如下。



slave 创建与 master 的连接后，首先发送 ping 指令，如果 master 没有返回异常，而是返回 pong，则说明 master 可用。如果 Redis 设置了密码，slave 会发送 `auth $masterauth` 指令，进行鉴权。当鉴权完毕，从库就通过 replconf 发送自己的端口及 IP 给 master。接下来，slave 继续通过 replconf 发送 `capa eof capa psync2` 进行复制版本校验。如果 master 校验成功。从库接下来就通过 psync 将自己的复制 id、复制偏移发送给 master，正式开始准备数据同步。

主库接收到从库发来的 psync 指令后，则开始判断可以进行数据同步的方式。前面讲到，Redis 当前保存了复制 id，replid 和 replid2。如果从库发来的复制 id，与 master 的复制 id（即 replid 和 replid2）相同，并且复制偏移在复制缓冲积压中，则可以进行增量同步。master 发送 continue 响应，并返回 master 的 replid。slave 将 master 的 replid 替换为自己的 replid，并将之前的复制 id 设置为 replid2。之后，master 则可继续发送，复制偏移位置 之后的指令，给 slave，完成数据同步。

如果主库发现从库传来的复制 id 和自己的 replid、replid2 都不同，或者复制偏移不在复制积压缓冲中，则判定需要进行全量复制。master 发送 fullresync 响应，附带 replid 及复制偏移。然后，master 根据需要构建 rdb，并将 rdb 及复制缓冲发送给 slave。

对于增量复制，slave 接下来就等待接受 master 传来的复制缓冲及新增的写指令，进行数据同步。

而对于全量同步，slave 会首先进行，嵌套复制的清理工作，比如 slave 当前还有嵌套的子slave，则该 slave 会关闭嵌套子slave的所有连接，并清理自己的复制积压缓冲。然后，slave 会构建临时 rdb 文件，并从 master 连接中读取 rdb 的实际数据，写入 rdb 中。在写 rdb 文件时，每写 8M，就会做一个 fsync 操作，刷新文件缓冲。当接受 rdb 完毕则将 rdb 临时文件改名为 rdb 的真正名字。

接下来，slave 会首先清空老数据，即删除本地所有 DB 中的数据，并暂时停止从 master 继续接受数据。然后，slave 就开始全力加载 rdb 恢复数据，将数据从 rdb 加载到内存。在 rdb 加载完毕后，slave 重新利用与 master 的连接 socket，创建与 master 连接的 client，并在此注册读事件，可以开始接受 master 的写指令了。此时，slave 还会将 master 的 replid 和复制偏移设为自己的复制 id 和复制偏移 offset，并将自己的 replid2 清空，因为，slave 的所有嵌套子slave 接下来也需要进行全量复制。最后，slave 就会打开 aof 文件，在接受 master 的写指令后，执行完毕并写入到自己的 aof 中。

相比之前的 sync，psync2 优化很明显。在短时间断开连接、slave 重启、切主等多种场景，只要延迟不太久，复制偏移仍然在复制积压缓冲，均可进行增量同步。master 不用构建并发送巨大的 rdb，可以大大减轻 master 的负荷和网络带宽的开销。同时，slave 可以通过轻量的增量复制，实现数据同步，快速恢复服务，减少系统抖动。

但是，psync 依然严重依赖于复制缓冲积压，太大会占用过多内存，太小会导致频繁的全量复制。而且，由于内存限制，即便设置相对较大的复制缓冲区，在 slave 断开连接较久时，仍然很容易被复制缓冲积压冲刷，从而导致全量复制。

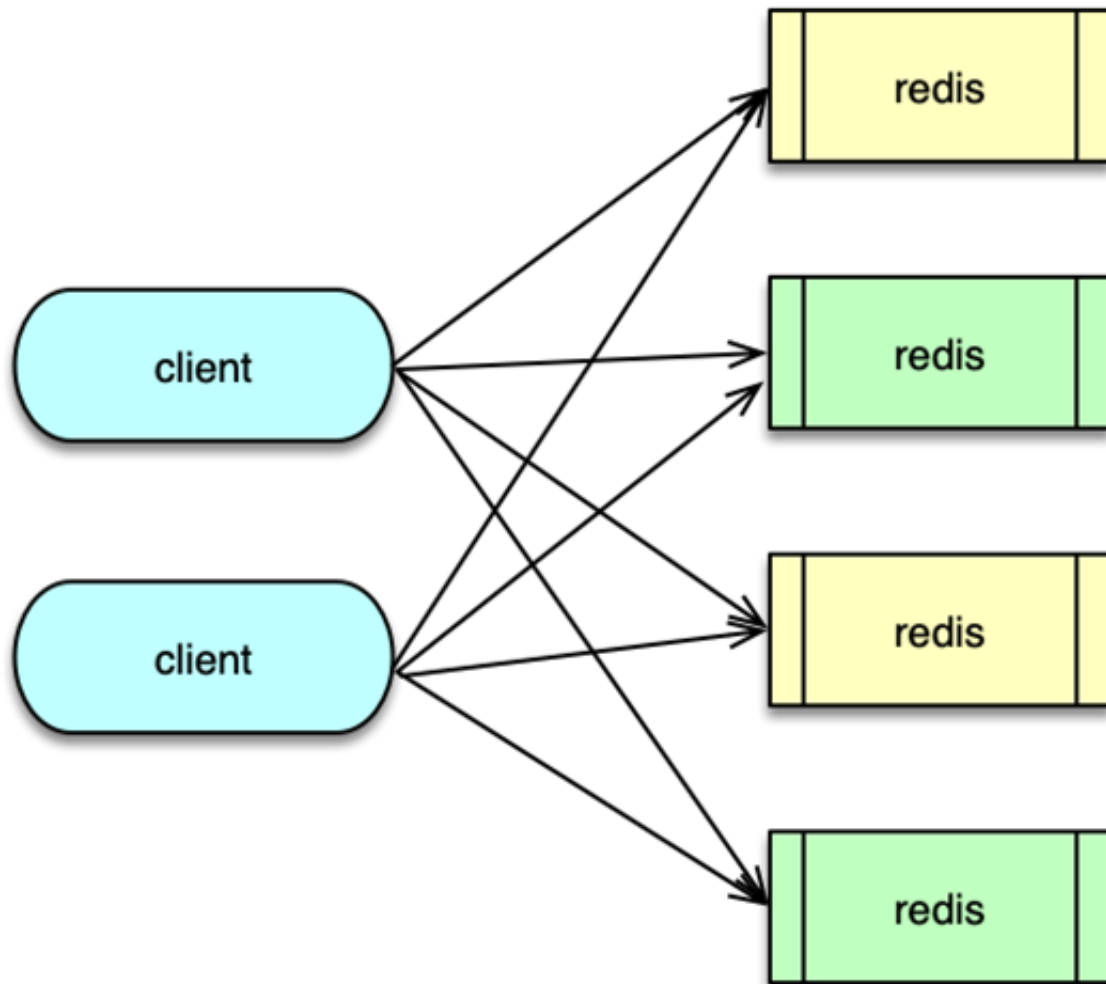
## 第28讲：如何构建一个高性能、易扩展的Redis集群？

通过上一课时的学习，我们知道复制功能可以 N 倍提升 Redis 节点的读性能，而集群则可以通过分布式方案来 N 倍提升 Redis 的写性能。除了提升性能之外，Redis 集群还可以提供更大的容量，提升资源系统的可用性。

Redis 集群的分布式方案主要有 3 种。分别是 Client 端分区方案，Proxy 分区方案，以及原生的 Redis Cluster 分区方案。

Client 端分区



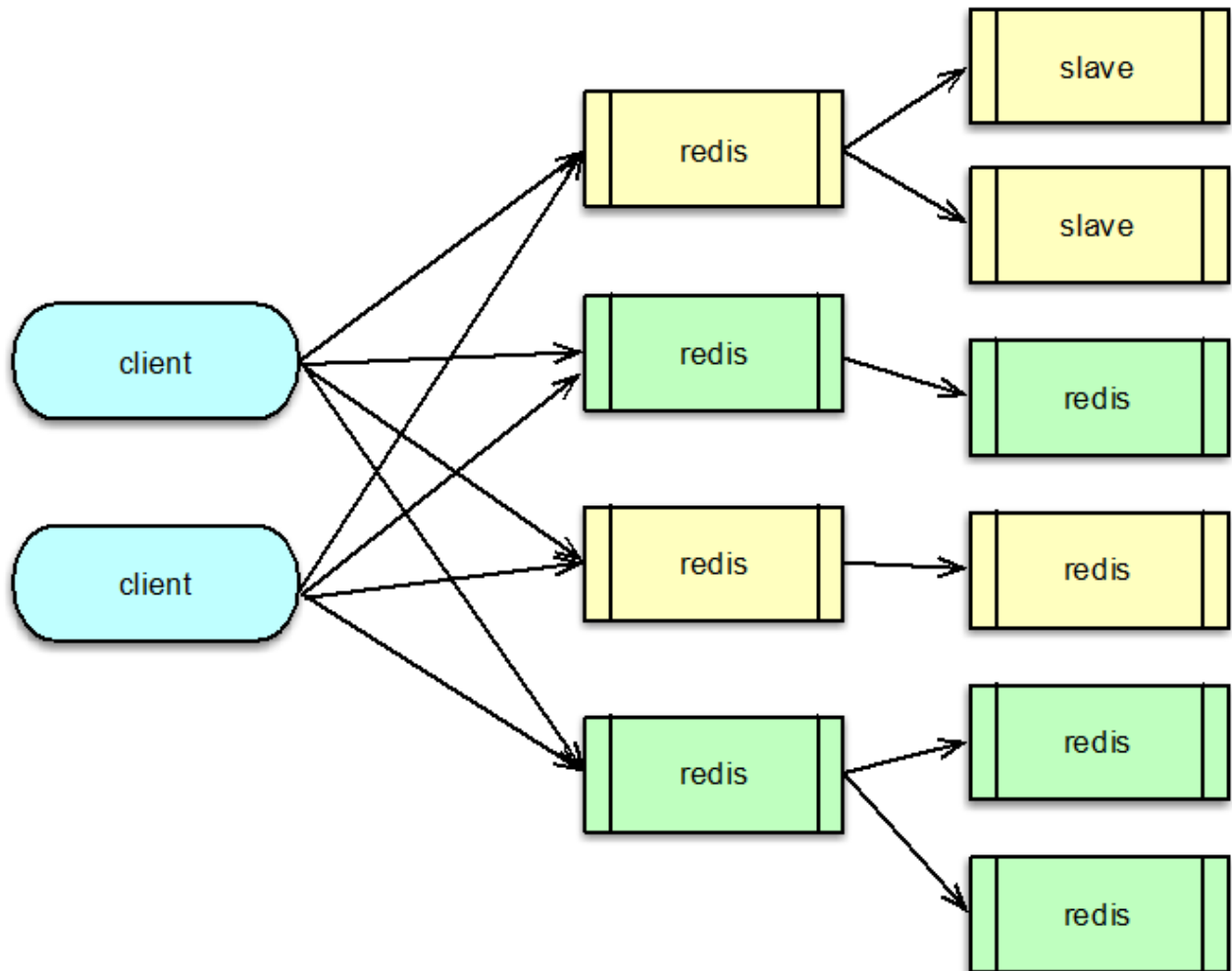


Client 端分区方案就是由 Client 决定数据被存储到哪个 Redis 分片，或者由哪个 Redis 分片来获取数据。它的核心思想是通过哈希算法将不同的 key 映射到固定的 Redis 分片节点上。对于单个 key 请求，Client 直接对 key 进行哈希后，确定 Redis 分片，然后进行请求。而对于一个请求附带多个 key 的场景，Client 会首先将这些 key 按哈希分片进行分类，从而将一个请求分拆为多个请求，然后再分别请求不同的哈希分片节点。

Client 通过哈希算法将数据进行分布，一般采用的哈希算法是取模哈希、一致性哈希和区间分布哈希。前两种哈希算法之前的课程已有详细分析，此处不在赘述。对于区间分布哈希，实际是一种取模哈希的变种，取模哈希是哈希并取模计算后，按哈希值来分配存储节点，而区间哈希是在哈希计算后，将哈希划分为多个区间，然后将这些区间分配给存储节点。如哈希后分 1024 个哈希点，然后将 0~511 作为分片 1，将 512~1023 作为分片 2。

对于 Client 端分区，由于 Redis 集群有多个 master 分片，同时每个 master 下挂载多个 slave，每个 Redis 节点都有独立的 IP 和端口。如果 master 异常需要切换 master，或读压力过大需要扩展新的 slave，这些都会涉及集群存储节点的变更，需要 Client 端做连接切换。





为了避免 Client 频繁变更 IP 列表，可以采用 DNS 的方式来管理集群的主从。对 Redis 集群的每个分片的主和从均采用不同 DNS 域名。Client 通过域名解析的方式获取域名下的所有 IP，然后来访问集群节点。由于每个分片 master 下有多个 slave，Client 需要在多个 slave 之间做负载均衡。可以按照权重建立与 slave 之间的连接，然后访问时，轮询使用这些连接依次访问，即可实现按权重访问 slave 节点。

在 DNS 访问模式下，Client 需要异步定时探测主从域名，如果发现 IP 变更，及时与新节点建立连接，并关闭老连接。这样在主库故障需要切换时，或者从库需要增加减少时，任何分片的主从变化，只需运维或管理进程改一下 DNS 下的 IP 列表，业务 Client 端不需要做任何配置变更，即可正常切换访问。

Client 端分区方案的优点在于分区逻辑简单，配置简单，Client 节点之间和 Redis 节点之间均无需协调，灵活性强。而且 Client 直接访问对应 Redis 节点，没有额外环节，性能高效。但该方案扩展不便。在 Redis 端，只能成倍扩展，或者预先分配足够多的分片。在 Client 端，每次分片后，业务端需要修改分发逻辑，并进行重启。

#### Proxy 端分区

Proxy 端分区方案是指 Client 发送请求给 Proxy 请求代理组件，Proxy 解析 Client 请求，并将请求分发到正确的 Redis 节点，然后等待 Redis 响应，最后再将结果返回给 Client 端。

如果一个请求包含多个 key，Proxy 需要将请求的多个 key，按分片逻辑分拆为多个请求，然后分别请求不同的 Redis 分片，接下来等待 Redis 响应，在所有的分拆响应到达后，再进行聚合组装，最后返回给 Client。在整个处理过程中，Proxy 代理首先要负责接受请求并解析，然后还要对 key 进行哈希计算及请求路由，最后还要将结果进行读取、解析及组装。如果系统运行中，主从变更或发生扩缩容，也只需由 Proxy 变更完成，业务 Client 端基本不受影响。

常见的 Proxy 端分区方案有 2 种，第一种是基于 Twemproxy 的简单分区方案，第二种是基于 Codis 的可平滑数据迁移的分区方案。

Twemproxy 是 Twitter 开源的一个组件，支持 Redis 和 Memcached 协议访问的代理组件。在讲分布式 Memcached 实战时，我曾经详细介绍了它的原理和实现架构，此处不再赘述。总体而言，Twemproxy 实现简单、稳定性高，在一些访问量不大且很少发生扩缩的业务场景中，可以很好的满足需要。但由于 Twemproxy 是单进程单线程模型的，对包含多个 key 的 mutli 请求，由于需要分拆请求，然后再等待聚合，处理性能较低。而且，在后端 Redis 资源扩缩容，即增加或减少分片时，需要修改配置并重启，无法做到平滑扩缩。而且 Twemproxy 方案默认只有一个代理组件，无管理后端，各种运维变更不够便利。

而 Codis 是一个较为成熟的分布式 Redis 解决方案。对于业务 Client 访问，连接 Codis-proxy 和连接单个 Redis 几乎没有区别。Codis 底层除了会自动解析分发请求之外，还可以在线进行数据迁移，使用非常方便。

Codis 系统主要由 Codis-server、Codis-proxy、Codis-dashboard、Zookeeper 等组成。

Codis-server 是 Codis 的存储组件，它是基于 Redis 的扩展，增加了 slot 支持和数据迁移功能，所有数据存储在预分配的 1024 个 slot 中，可以按 slot 进行同步或异步数据迁移。

Codis-proxy 处理 Client 请求，解析业务请求，并路由给后端的 Codis-server group。Codis 的每个 server group 相当于一个 Redis 分片，由 1 个 master 和 N 个从库组成。

Zookeeper 用于存储元数据，如 Proxy 的节点，以及数据访问的路由表。除了 Zookeeper，Codis 也支持 etcd 等其他组件，用于元数据的存储和通知。

Codis-dashboard 是 Codis 的管理后台，可用于管理数据节点、Proxy 节点的加入或删除，还可用于执行数据迁移等操作。Dashboard 的各项变更指令通过 Zookeeper 进行分发。

Codis 提供了功能较为丰富的管理后台，可以方便的对整个集群进行监控及运维。

Proxy 端分区方案的优势，是 Client 访问逻辑和 Redis 分布逻辑解耦，业务访问便捷简单。在资源发生变更或扩缩容时，只用修改数量有限的 Proxy 即可，数量庞大的业务 Client 端不用做调整。

但 Proxy 端分区的方案，访问时请求需要经过 Proxy 中转，访问多跳了一级，性能会存在损耗，一般损耗会达到 5~15% 左右。另外多了一个代理层，整个系统架构也会更复杂。

Redis Cluster 分区

Redis 社区版在 3.0 后开始引入 Cluster 策略，一般称之为 Redis-Cluster 方案。Redis-Cluster 按 slot 进行数据的读写和管理，一个 Redis-Cluster 集群包含 16384 个 slot。每个 Redis 分片负责其中一部分 slot。在集群启动时，按需将所有 slot 分配到不同节点，在集群系统运行后，按 slot 分配策略，将 key 进行 hash 计算，并路由到对应节点 访问。

随着业务访问模型的变化，Redis 部分节点可能会出现压力过大、访问不均衡的现象，此时可以将 slot 在 Redis 分片节点内部进行迁移，以均衡访问。如果业务不断发展，数据量过大、TPS 过高，还可以将 Redis 节点的部分 slot 迁移到新节点，增加 Redis-Cluster 的分片，对整个 Redis 资源进行扩容，已提升整个集群的容量及读写能力。

在启动 Redis 集群时，在接入数据读写前，可以通过 Redis 的 Cluster addslots 将 16384 个 slot 分配给不同的 Redis 分片节点，同时可以用 Cluster delslots 去掉某个节点的 slot，用 Cluster flushslots 清空某个节点的所有 slot 信息，来完成 slot 的调整。

Redis Cluster 是一个去中心化架构，每个节点记录全部 slot 的拓扑分布。这样 Client 如果把 key 分发给了错误的 Redis 节点，Redis 会检查请求 key 所属的 slot，如果发现 key 属于其他节点的 slot，会通知 Client 重定向到正确的 Redis 节点访问。

Redis Cluster 下的不同 Redis 分片节点通过 gossip 协议进行互联，使用 gossip 的优势在于，该方案无中心控制节点，这样，更新不会受到中心节点的影响，可以通过通知任意一个节点来进行管理通知。不足就是元数据的更新会有延时，集群操作会在一定的时延后才会通知到所有 Redis。由于 Redis Cluster 采用 gossip 协议进行服务节点通信，所以在进行扩缩容时，可以向集群内任何一个节点，发送 Cluster meet 指令，将新节点加入集群，然后集群节点会立即扩散新节点，到整个集群。meet 新节点操作的扩散，只需要有一条节点链能到达集群各个节点即可，无需 meet 所有集群节点，操作起来比较便利。

在 Redis-Cluster 集群中，key 的访问需要 smart client 配合。Client 首先发送请求给 Redis 节点，Redis 在接受并解析命令后，会对 key 进行 hash 计算以确定 slot 槽位。计算公式是对 key 做 crc16 哈希，然后对 16383 进行按位与操作。如果 Redis 发现 key 对应的 slot 在本地，则直接执行后返回结果。

如果 Redis 发现 key 对应的 slot 不在本地，会返回 moved 异常响应，并附带 key 的 slot，以及该 slot 对应的正确 Redis 节点的 host 和 port。Client 根据响应解析出正确的节点 IP 和端口，然后把请求重定向到正确的 Redis，即可完成请求。为了加速访问，Client 需要缓存 slot 与 Redis 节点的对应关系，这样可以直接访问正确的节点，以加速访问性能。

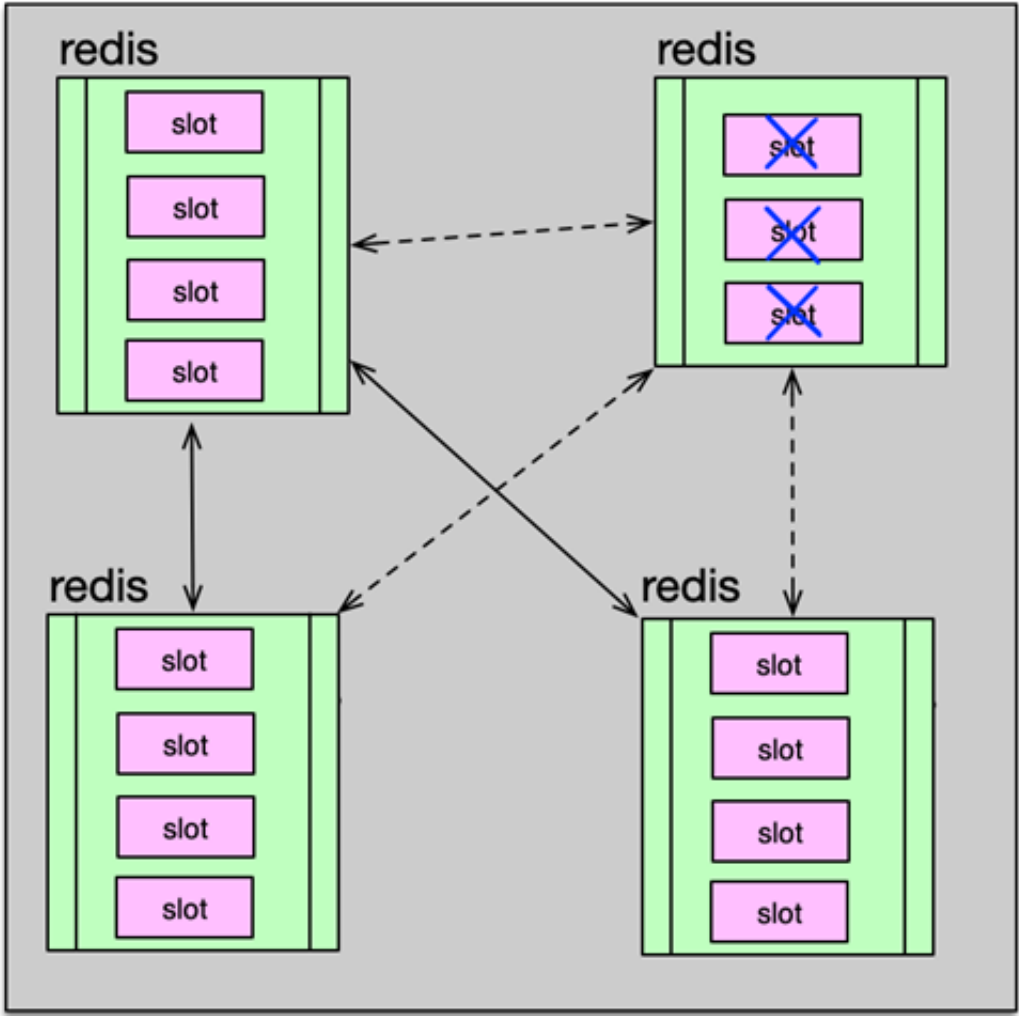
Redis-Cluster 提供了灵活的节点扩缩容方案，可以在不影响用户访问的情况下，动态为集群增加节点扩容，或下线节点为集群缩容。由于扩容在线上最为常见，我首先来分析一下 Redis-Cluster 如何进行扩容操作。

在准备对 Redis 扩容时，首先准备待添加的新节点，部署 Redis，配置 cluster-enable 为 true，并启动。然后运维人员，通过 client 连接上一个集群内的 Redis 节点，通过 cluster meet 命令将新节点加入到集群，该节点随后会通知集群内的其他节点，有新节点加入。因为新加入的节点还没有设置任何 slot，所以不接受任何读写操作。

然后，将通过 `cluster setslot $slot importing` 指令，在新节点中，将目标 slot 设为 importing 导入状态。再将 slot 对应的源节点，通过 `cluster setslot $slot migrating` 将源节点的 slot 设为 migrating 迁移导出状态。

接下来，就从源节点获取待迁移 slot 的 key，通过 `cluster getkeysinslot $slot $count` 命令，从 slot 中获取 N 个待迁移的 key。然后通过 `migrate` 指令，将这些 key 依次逐个迁移或批量一次迁移到目标新节点。对于迁移单个 key，使用指令 `migrate $host $port $key $dbid timeout`，如果一次迁移多个 key，在指令结尾加上 `keys` 选项，同时将多个 key 放在指令结尾即可。持续循环前面 2 个步骤，不断获取 slot 里的 key，然后进行迁移，最终将 slot 下的所有数据都迁移到目标新节点。最后通过 `cluster setslot` 指令将这个 slot 指派给新增节点。`setslot` 指令可以发给集群内的任意一个节点，这个节点会将这个指派信息扩散到整个集群。至此，slot 就迁移到了新节点。如果要迁移多个 slot，可以继续前面的迁移步骤，最终将所有需要迁移的 slot 数据搬到新节点。

这个新迁移 slot 的节点属于主库，对于线上应用，还需要增加从库，以增加读写能力及可用性，否则一旦主库崩溃，整个分片的数据就无法访问。在节点上增加从库，需要注意的是，不能使用非集群模式下的 `slaveof` 指令，而要使用 `cluster replication`，才能完成集群分片节点下的 slave 添加。另外，对于集群模式，slave 只能挂在分片 master 上，slave 节点自身不能再挂载 slave。

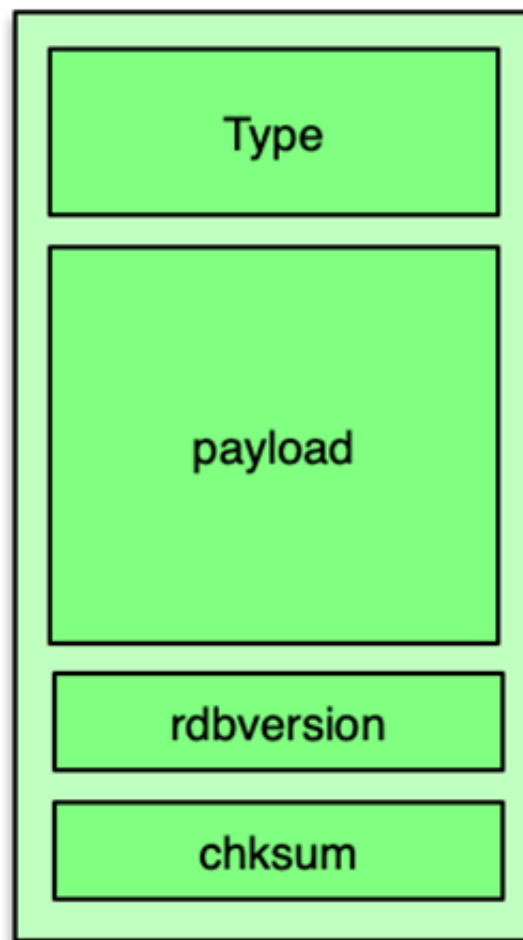


缩容流程与扩容流程类似，只是把部分节点的 slot 全部迁移走，然后把这些没有 slot 的节点进行下线处理。在下线老节点之前，需要注意，要用 `cluster forget` 通知集群，集群节点要，从节点信息列表中，将目标节点移除，同时会将该节点加入到禁止列表，1 分钟之内不允许再加入集群。以防止在扩散下线节点时，又被误加入集群。

Redis 社区官方在源代码中也提供了 redis-trib.rb，作为 Redis Cluster 的管理工具。该工具用 Ruby 开发，所以在使用前，需要安装相关的依赖环境。redis-trib 工具通过封装前面所述的 Redis 指令，从而支持创建集群、检查集群、添加删除节点、在线迁移 slot 等各种功能。

Redis Cluster 在 slot 迁移过程中，获取key指令以及迁移指令逐一发送并执行，不影响 Client 的正常访问。但在迁移单条或多条 key 时，Redis 节点是在阻塞状态下进行的，也就是说，Redis 在迁移 key 时，一旦开始执行迁移指令，就会阻塞，直到迁移成功或确认失败后，才会停止该 key 的迁移，从而继续处理其他请求。slot 内的 key 迁移是通过 migrate 指令进行的。

在源节点接收到 migrate \$host \$port \$key \$destination-db 的指令后，首先 slot 迁移的源节点会与迁移的目标节点建立 socket 连接，第一次迁移，或者迁移过程中，当前待迁移的 DB 与前一次迁移的 DB 不同，在迁移数据前，还需要发送 select \$dbid 进行切换到正确的 DB。



然后，源节点会轮询所有待迁移的 key/value。获取 key 的过期时间，并将 value 进行序列化，序列化过程就是将 value 进行 dump，转换为类 rdb 存储的二进制格式。这个二进制格式分 3 部分。第一部分是 value 对象的 type。第二部分是 value 实际的二进制数据；第三部分是当前 rdb 格式的版本，以及该 value 的 CRC64 校验码。至此，待迁移发送的数据准备完毕，源节点向目标节点，发送 restore-asking 指令，将过期时间、key、value 的二进制数据发送给目标节点。然后同步等待目标节点的响应结果。

目标节点对应的client，收到指令后，如果有 select 指令，就首先切换到正确的 DB。接下来读取并处理 restore-asking 指令，处理 restore-asking 指令时，首先对收到的数据进行解析校验，获取 key 的 ttl，校验 rdb 版本及 value 数据 crc64 校验码，确认无误后，将数据存入 redisDb，设置过期时间，并返回响应。

源节点收到目标节点处理成功的响应后。对于非 copy 类型的 migrate，会删除已迁移的 key。至此，key 的迁移就完成了。migrate 迁移指令，可以一次迁移一个或多个 key。注意，整个迁移过程中，源节点在发送 restore-asking 指令后，同步阻塞，等待目标节点完成数据处理，直到超时或者目标节点返回响应结果，收到结果后在本地处理完毕后序事件，才会停止阻塞，才能继续处理其他事件。所以，单次迁移的 key 不能太多，否则阻塞时间会较长，导致 Redis 卡顿。同时，即便单次只迁移一个 key，如果对应的 value 太大，也可能导致 Redis 短暂卡顿。

在 slot 迁移过程中，不仅其他非迁移 slot 的 key 可以正常访问，即便正在迁移的 slot，它里面的 key 也可以正常读写，不影响业务访问。但由于 key 的迁移是阻塞模式，即在迁移 key 的过程中，源节点并不会处理任何请求，所以在 slot 迁移过程中，待读写的 key 只有三种存在状态。

尚未被迁移，后续会被迁走；

已经被迁移；

这个 key 之前并不存在集群中，是一个新 key。

slot 迁移过程中，对节点里的 key 处理方式如下。

对于尚未被迁移的 key，即从 DB 中找到该 key，不管这个 key 所属的 slot 是否正在被迁移，都直接在本地进行读写处理。

对于无法从 DB 中找到 value 的 key，但key所属slot正在被迁移，包括已迁走或者本来不存在的 key 两种状态，Redis 返回 ask 错误响应，并附带 slot 迁移目标节点的 host 和 port。Client 收到 ask 响应后，将请求重定向到 slot 迁移的新节点，完成响应处理。

对于无法从 DB 中找到 value 的 key，且 key 所在的 slot 不属于本节点，说明 Client 发送节点有误，直接返回 moved 错误响应，也附上 key 对应节点的 host 和 port，由 Client 重定向请求。

对于 Redis Cluster 集群方案，由社区官方实现，并有 Redis-trib 集群工具，上线和使用起来比较便捷。同时它支持在线扩缩，可以随时通过工具查看集群的状态。但这种方案也存在不少弊端。首先，数据存储和集群逻辑耦合，代码逻辑复杂，容易出错。

其次，Redis 节点要存储 slot 和 key 的映射关系，需要额外占用较多内存，特别是对 value size 比较小、而key相对较大的业务，影响更是明显。

再次，key 迁移过程是阻塞模式，迁移大 value 会导致服务卡顿。而且，迁移过程，先获取 key，再迁移，效率低。最后，Cluster 模式下，集群复制的 slave 只能挂载到 master，不支持 slave 嵌套，会导致 master 的压力过大，无法支持那些，需要特别多 slave、读 TPS 特别大的业务场景。

## 第29讲：从容应对亿级QPS访问，Redis还缺少什么？

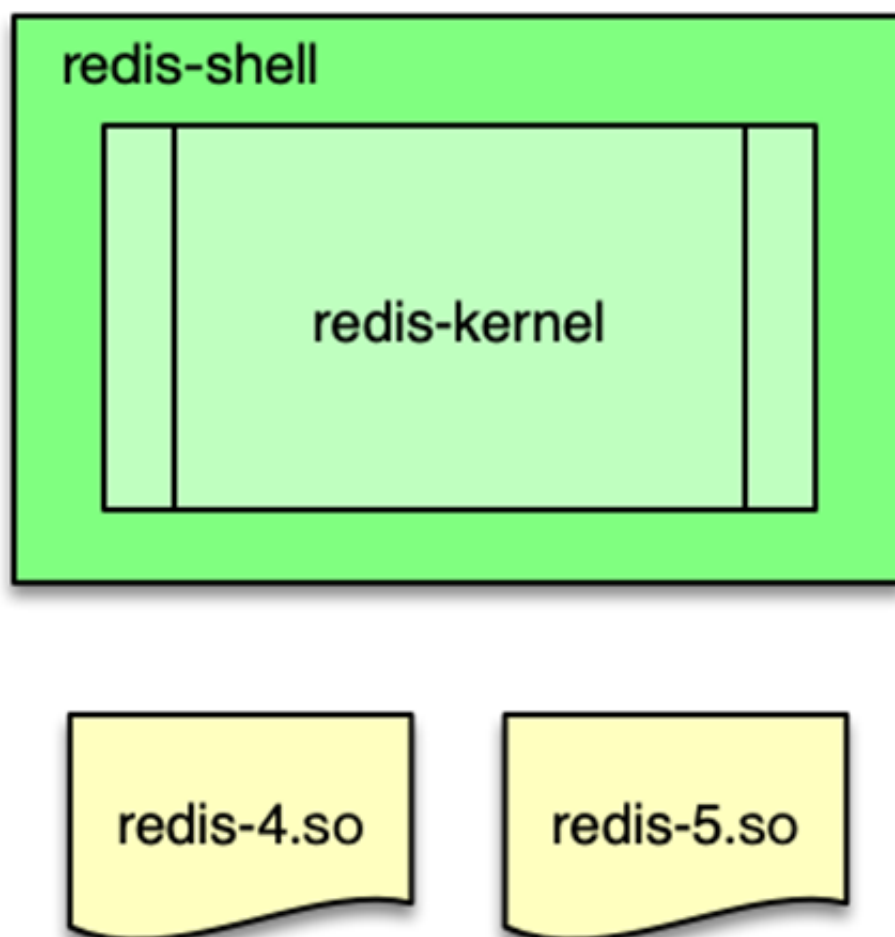
众所周知，Redis 在线上实际运行时，面对海量数据、高并发访问，会遇到不少问题，需要进行针对性扩展及优化。本课时，我会结合微博在使用 Redis 中遇到的问题，来分析如何在生产环境下对 Redis 进行扩展改造，以应对百万级 QPS。

## 功能扩展

对于线上较大流量的业务，单个 Redis 实例的内存占用很容易达到数 G 的容量，对应的 aof 会占用数十 G 的空间。即便每天流量低峰时间，对 Redis 进行 rewriteaof，减少数据冗余，但由于业务数据多，写操作多，aof 文件仍然会达到 10G 以上。

此时，在 Redis 需要升级版本或修复 bug 时，如果直接重启变更，由于需要数据恢复，这个过程需要近 10 分钟的时间，时间过长，会严重影响系统的可用性。面对这种问题，可以对 Redis 扩展热升级功能，从而在毫秒级完成升级操作，完全不影响业务访问。

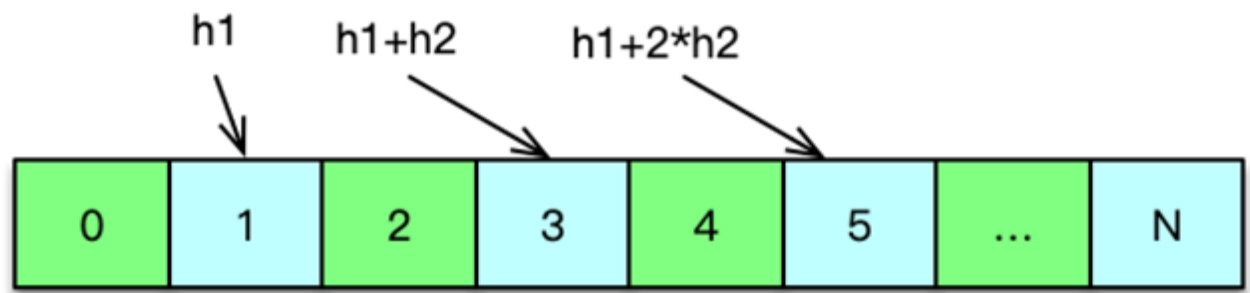
# redis



热升级方案如下，首先构建一个 Redis 壳程序，将 redisServer 的所有属性（包括redisDb、client等）保存为全局变量。然后将 Redis 的处理逻辑代码全部封装到动态连接库 so 文件中。Redis 第一次启动，从磁盘加载恢复数据，在后续升级时，通过指令，壳程序重新加载 Redis 新的 so 文件，即可完成功能升级，毫秒级完成 Redis 的版本升级。而且整个过程中，所有 Client 连接仍然保留，在升级成功后，原有 Client 可以继续进行读写操作，整个过程对业务完全透明。

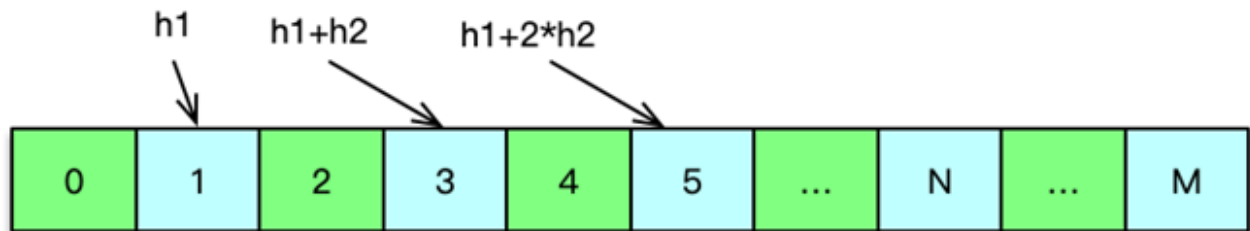
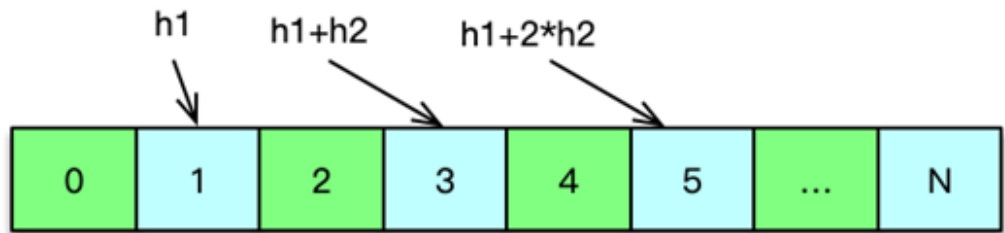


在 Redis 使用中，也经常会遇到一些特殊业务场景，是当前 Redis 的数据结构无法很好满足的。此时可以对 Redis 进行定制化扩展。可以根据业务数据特点，扩展新的数据结构，甚至扩展新的 Redis 存储模型，来提升 Redis 的内存效率和处理性能。



在微博中，有个业务类型是关注列表。关注列表存储的是一个用户所有关注的用户 uid。关注列表可以用来验证关注关系，也可以用关注列表，进一步获取所有关注人的微博列表等。由于用户数量过于庞大，存储关注列表的 Redis 是作为一个缓存使用的，即不活跃的关注列表会很快被踢出 Redis。在再次需要这个用户的关注列表时，重新从 DB 加载，并写回 Redis。关注列表的元素全部 long，最初使用 set 存储，回种 set 时，使用 sadd 进行批量添加。线上发现，对于关注数比较多的关注列表，比如关注数有数千上万个用户，需要 sadd 上成千上万个 uid，即便分几次进行批量添加，每次也会消耗较多时间，数据回种效率较低，而且会导致 Redis 卡顿。另外，用 set 存关注列表，内存效率也比较低。

于是，我们对 Redis 扩展了 longset 数据结构。longset 本质上是一个 long 型的一维开放数组。可以采用 double-hash 进行寻址。

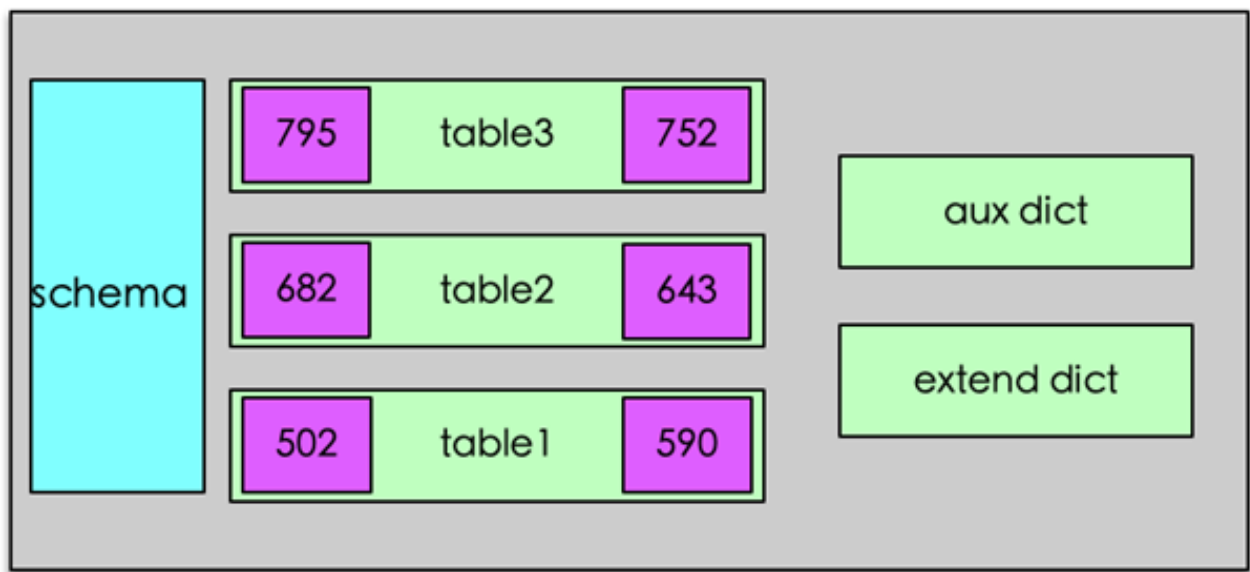


从 DB 加载到用户的关注列表，准备写入 Redis 前。Client 首先根据关注的 uid 列表，构建成 long 数组的二进制格式，然后通过扩展的 lisset 指令写入 Redis。Redis 接收到指令后，直接将 Client 发来的二进制格式的 long 数组作为 value 值进行存储。

longset 中的 long 数组，采用 double-hash 进行寻址，即对每个 long 值采用 2 个哈希函数计算，然后按  $(h1 + n \cdot h2) \% \text{数组长度}$  的方式，确定 long 值的位置。n 从 0 开始计算，如果出现哈希冲突，即计算的哈希位置，已经有其他元素，则 n 加 1，继续向前推进计算，最大计算次数是数组的长度。



在向 longset 数据结构不断增加 long 值元素的过程中，当数组的填充率超过阈值，Redis 则返回 longset 过满的异常。此时 Client 会根据最新全量数据，构建一个容量加倍的一维 long 数组，再次 lset 回 Redis 中。

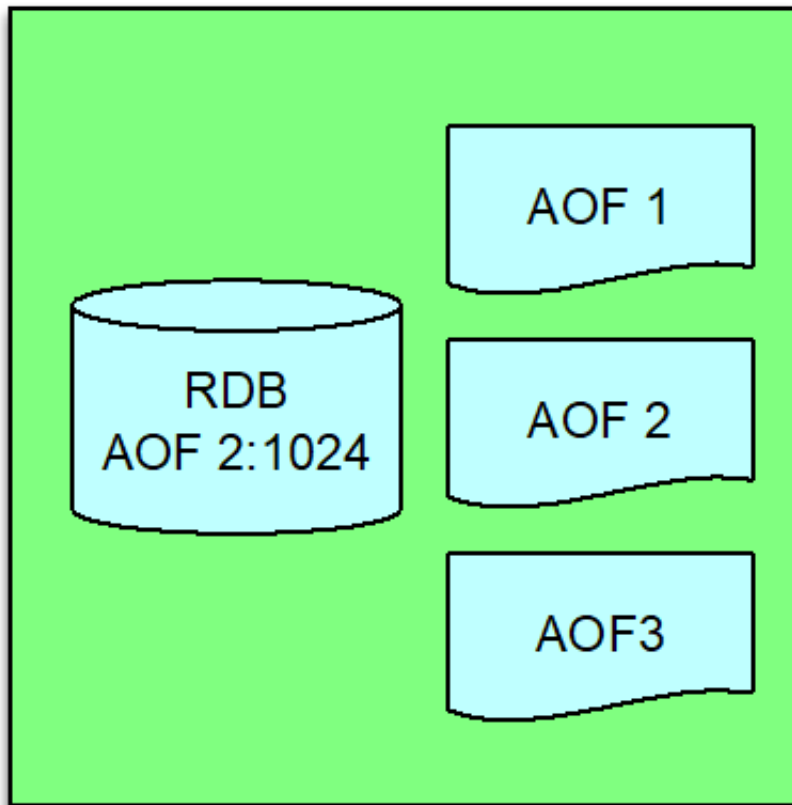


在移动社交平台中，庞大的用户群体，相互之间会关注、订阅，用户自己会持续分享各种状态，另外这些状态数据会被其他用户阅读、评论、扩散及点赞。这样，在用户维度，就有关注数、粉丝数、各种状态行为数，然后用户每发表的一条 feed、状态，还有阅读数、评论数、转发数、表态数等。一方面会有海量 key 需要进行计数，另外一方面，一个 key 会有 N 个计数。在日常访问中，一次查询，不仅需要查询大量的 key，而且对每个 key 需要查询多个计数。

以微博为例，历史计数高达千亿级，而且随着每日新增数亿条 feed 记录，每条记录会产生 4~8 种计数，如果采用 Redis 的计数，仅仅单副本存储，历史数据需要占用 5~6T 以上的内存，每日新增 50G 以上，如果再考虑多 IDC、每个 IDC 部署 1 主多从，占用内存还要再提升一个数量级。由于微博计数，所有的 key 都是随时间递增的 long 型值，于是我们改造了 Redis 的存储结构。

首先采用 cdb 分段存储计数器，通过预先分配的内存数组 Table 存储计数，并且采用 double hash 解决冲突，避免 Redis 实现中的大量指针开销。然后，通过 Schema 策略支持多列，一个 key id 对应的多个计数可以作为一条计数记录，还支持动态增减计数列，每列的计数内存使用精简到 bit。而且，由于 feed 计数冷热区分明显，我们进行冷热数据分离存储方案，根据时间维度，近期的热数据放在内存，之前的冷数据放在磁盘，降低机器成本。

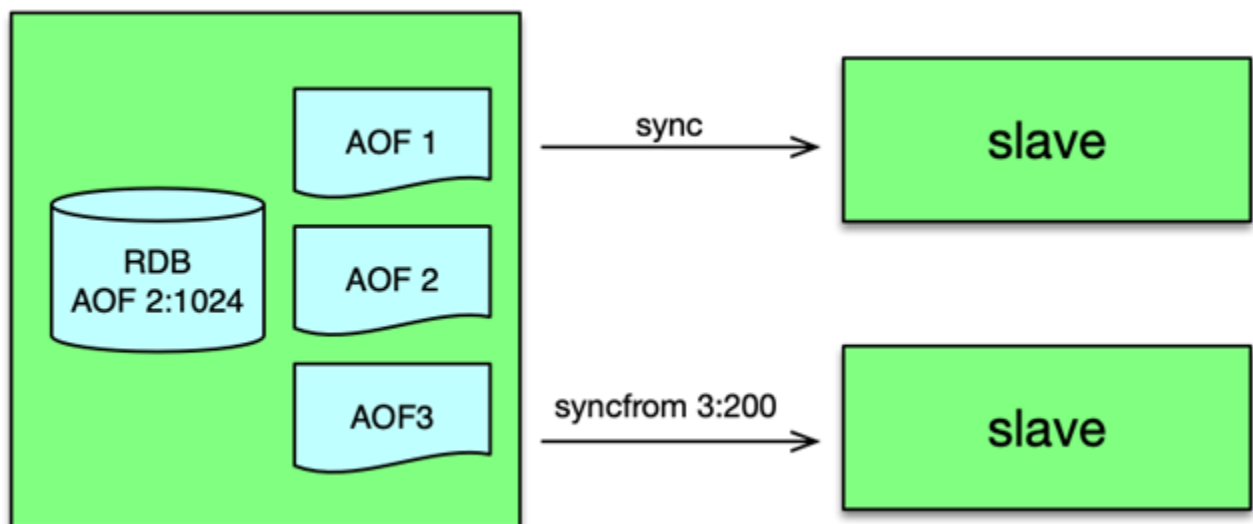
关于计数器服务的扩展，后面的案例分析课时，我会进一步深入介绍改造方案。



线上 Redis 使用，不管是最初的 sync 机制，还是后来的 psync 和 psync2，主从复制都会受限于复制积压缓冲。如果 slave 断开复制连接的时间较长，或者 master 某段时间写入量过大，而 slave 的复制延迟较大，slave 的复制偏移量落在 master 的复制积压缓冲之外，则会导致全量复制。

完全增量复制

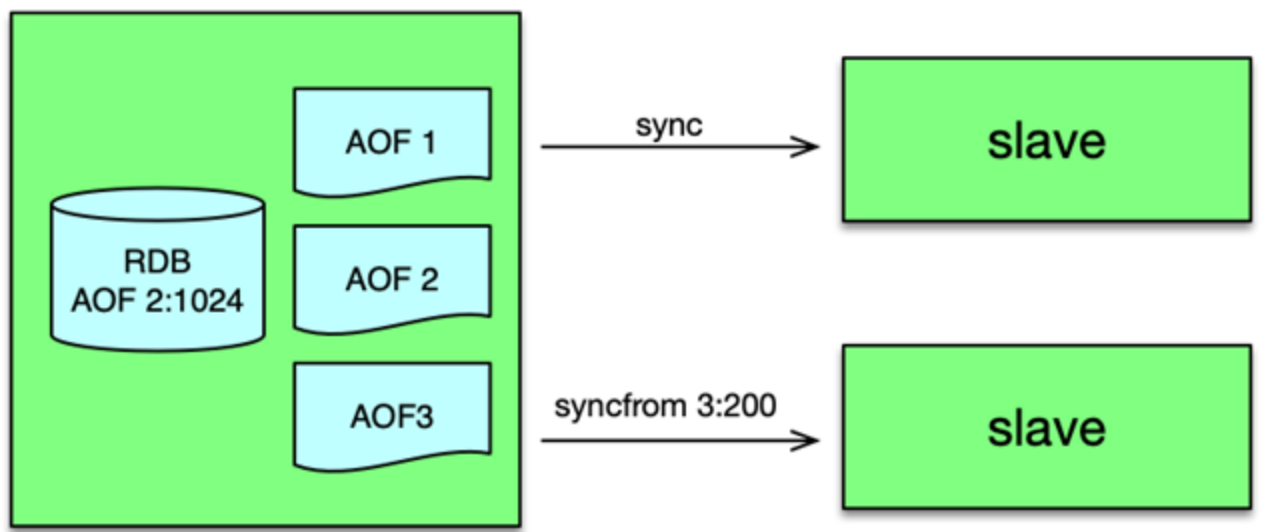
于是，微博整合 Redis 的 rdb 和 aof 策略，构建了完全增量复制方案。



在完全增量方案中，aof 文件不再只有一个，而是按后缀 id 进行递增，如 aof.00001、aof.00002，当 aof 文件超过阈值，则创建下一个 id 加 1 的文件，从而滚动存储最新的写指令。在 bgsave 构建 rdb 时，rdb 文件除了记录当前的内存数据快照，还会记录 rdb 构建时间，对应 aof 文件的 id 及位置。这样 rdb 文件和其记录 aof 文件位置之后的写指令，就构成一份完整的最新数据记录。

主从复制时，master 通过独立的复制线程向 slave 同步数据。每个 slave 会创建一个复制线程。第一次复制是全量复制，之后的复制，不管 slave 断开复制连接有多久，只要 aof 文件没有被删除，都是增量复制。

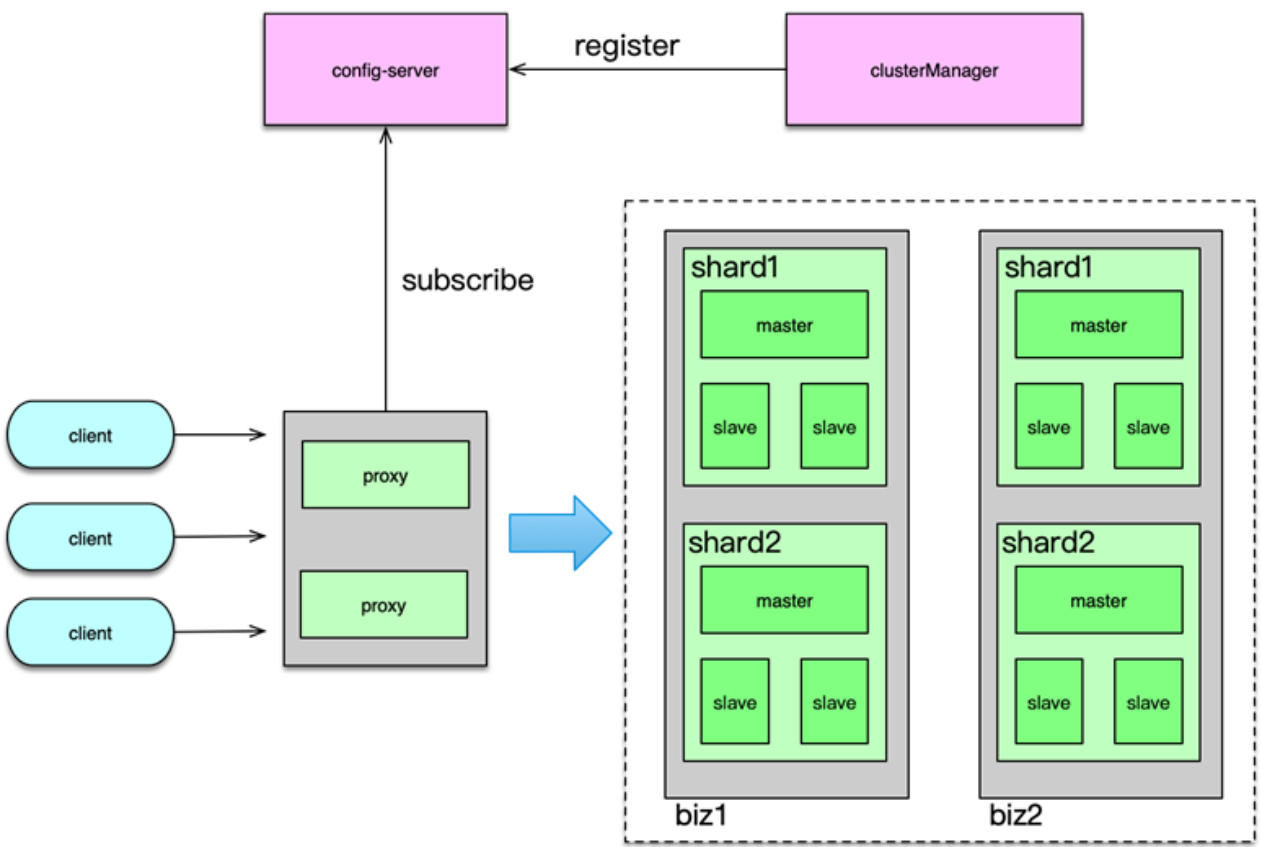
第一次全量复制时，复制线程首先将 rdb 发给 slave，然后再将 rdb 记录的 aof 文件位置之后的所有数据，也发送给 slave，即可完成。整个过程不用重新构建 rdb。



后续同步时，slave 首先传递之前复制的 aof 文件的 id 及位置。master 的复制线程根据这个信息，读取对应 aof 文件位置之后的所有内容，发送给 slave，即可完成数据同步。

由于整个复制过程，master 在独立复制线程中进行，所以复制过程不影响用户的正常请求。为了减轻 master 的复制压力，全增量复制方案仍然支持 slave 嵌套，即可以在 slave 后继续挂载多个 slave，从而把复制压力分散到多个不同的 Redis 实例。

集群管理



前面讲到，Redis-Cluster 的数据存储和集群逻辑耦合，代码逻辑复杂易错，存储 slot 和 key 的映射需要额外占用较多内存，对小 value 业务影响特别明显，而且迁移效率低，迁移大 value 容易导致阻塞，另外，Cluster 复制只支持 slave 挂在 master 下，无法支持 需要较多slave、读 TPS 特别大的业务场景。除此之外，Redis 当前还只是个存储组件，线上运行中，集群管理、日常维护、状态监控报警等这些功能，要么没有支持，要么支持不便。

因此我们也基于 Redis 构建了集群存储体系。首先将 Redis 的集群功能剥离到独立系统，Redis 只关注存储，不再维护 slot 等相关的信息。通过新构建的 clusterManager 组件，负责 slot 维护，数据迁移，服务状态管理。

Redis 集群访问可以由 proxy 或 smart client 进行。对性能特别敏感的业务，可以通过 smart client 访问，避免访问多一跳。而一般业务，可以通过 Proxy 访问 Redis。

业务资源的部署、Proxy 的访问，都通过配置中心进行获取及协调。clusterManager 向配置中心注册业务资源部署，并持续探测服务状态，根据服务状态进行故障转移，切主、上下线 slave 等。proxy 和 smart client 从配置中心获取配置信息，并持续订阅服务状态的变化。

## 第十章：深入分布式缓存

---

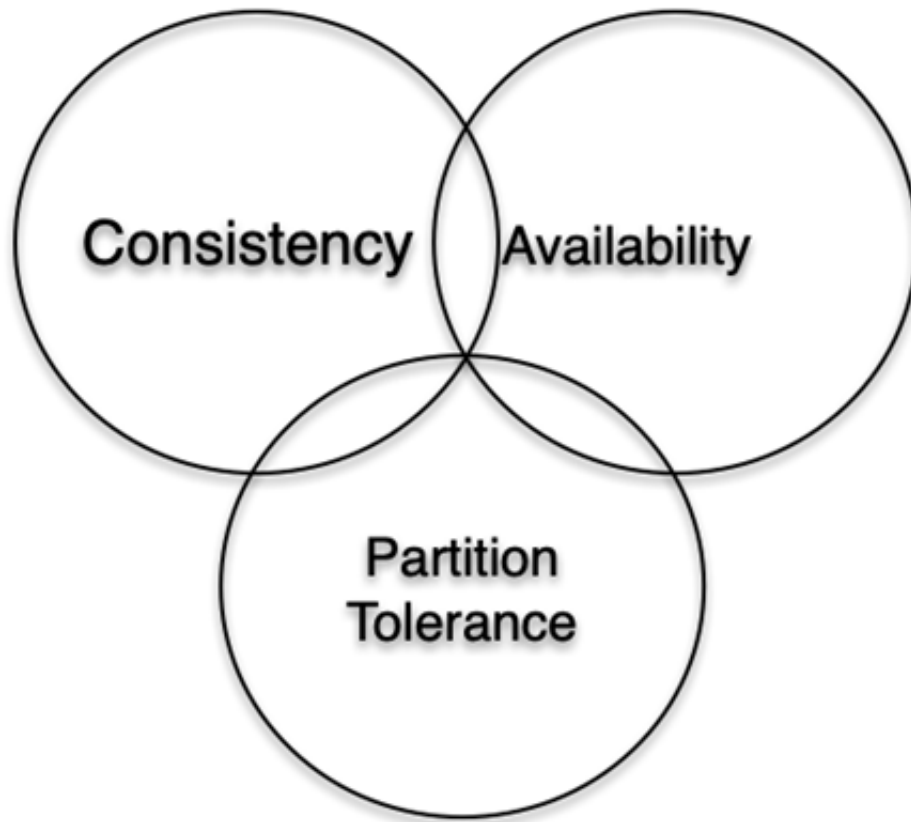
### 第30讲：面对海量数据，为什么无法设计出完美的分布式缓存体系？

随着互联网的发展，分布式系统变得越来越重要，当前的大中型互联网系统几乎都向着分布式方向发展。分布式系统简单说就是一个软硬件分布在不同机房、不同区域的网络计算机上，彼此之间仅仅通过消息传递进行通信及协调的系统。分布式系统需要利用分布的服务，在确保数据一致的基础上，对外提供稳定的服务。

#### CAP 定理的诞生

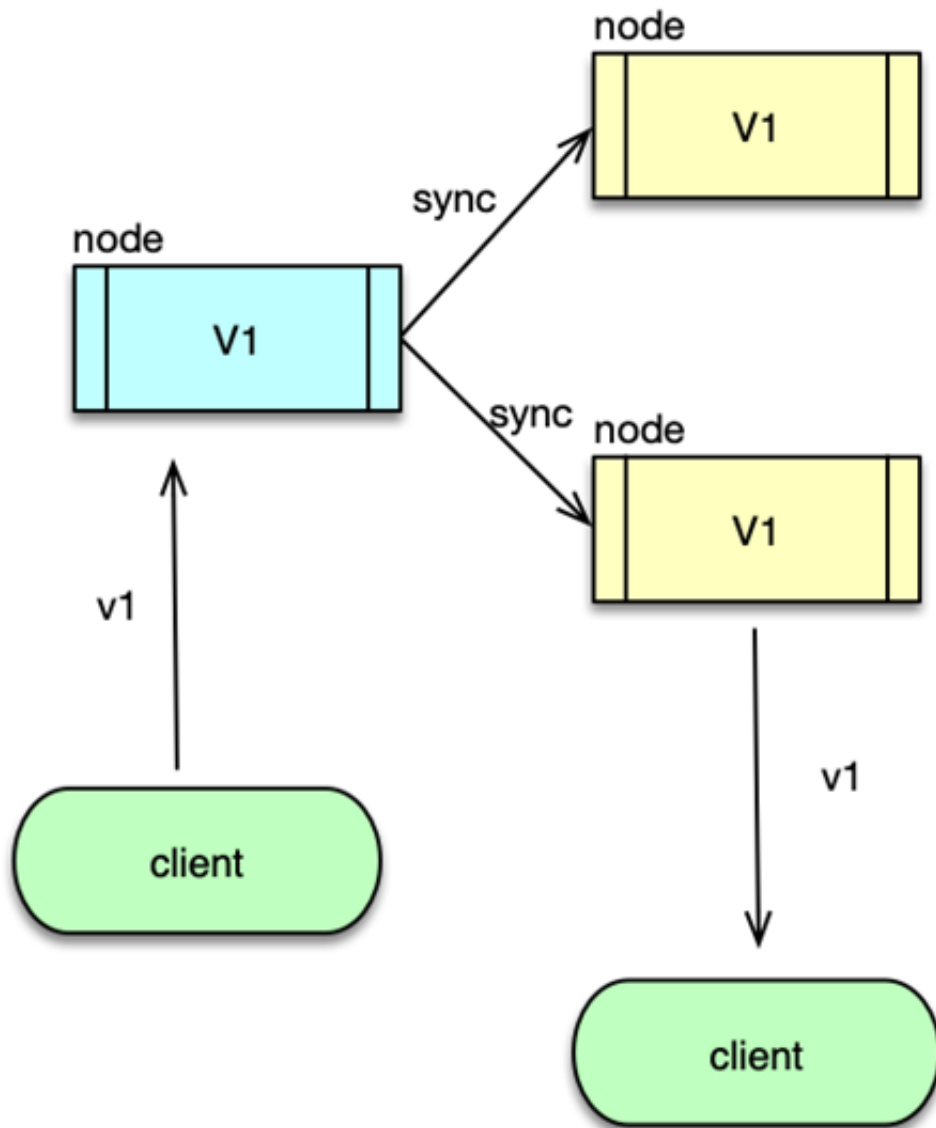
在分布式系统的发展中，影响最大最广泛的莫过于 CAP 理论了，可以说 CAP 理论是分布式系统发展的理论基石。早在 1998 年，加州大学的计算机科学家 Eric Brewer，就提出分布式系统的三个指标。在此基础上，2 年后，Eric Brewer 进一步提出了 CAP 猜想。又过了 2 年，到了 2002 年，麻省理工学院的 Seth Gilbert 和 Nancy Lynch 从理论上证明了 CAP 猜想。CAP 猜想成为了 CAP 定理，也称为布鲁尔定理。从此，CAP 定理成为分布式系统发展的理论基石，广泛而深远的影响着分布式系统的发展。

#### CAP 定理指标



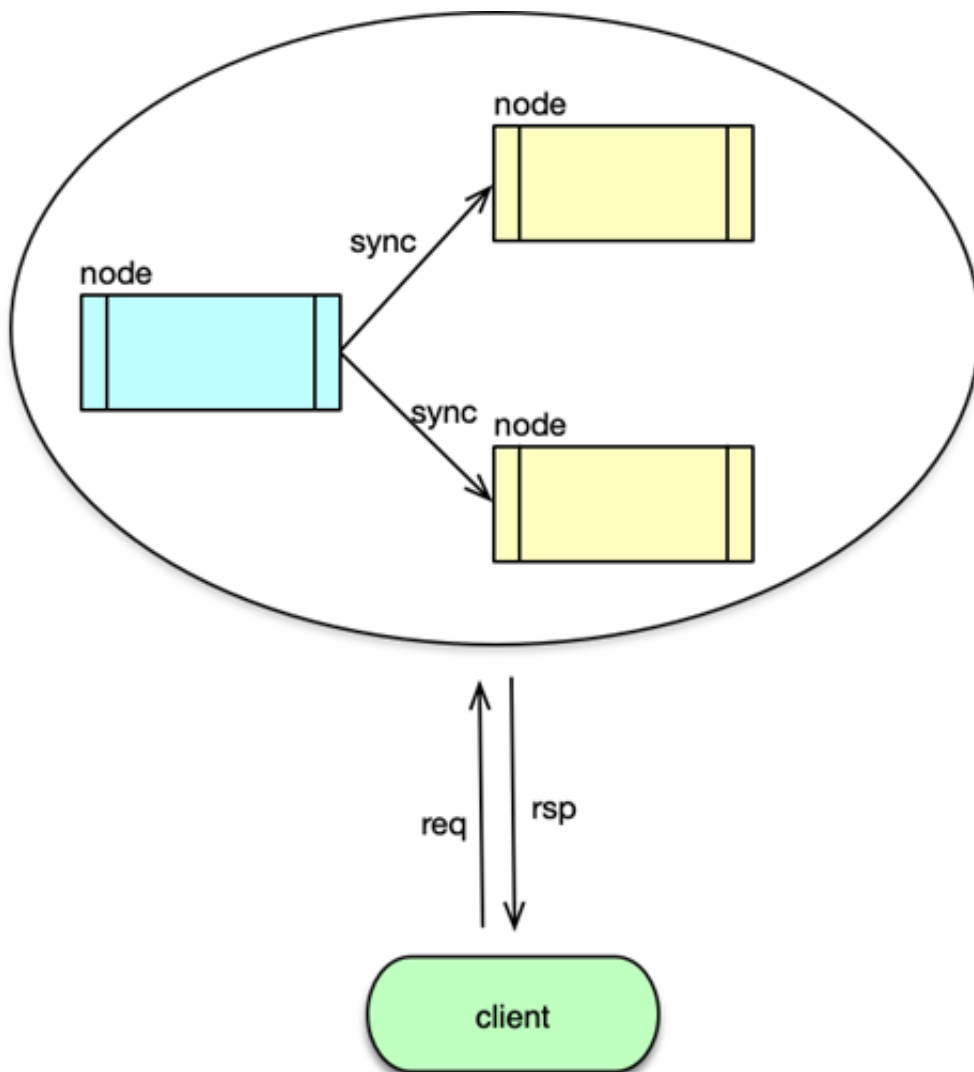
CAP 定理，简单的说就是分布式系统不可能同时满足 Consistency 一致性、Availability 可用性、Partition Tolerance 分区容错性三个要素。因为 Consistency、Availability 、Partition Tolerance 这三个单词的首字母分别是 C、A、P，所以这个结论被称为 CAP 定理。

Consistency 一致性



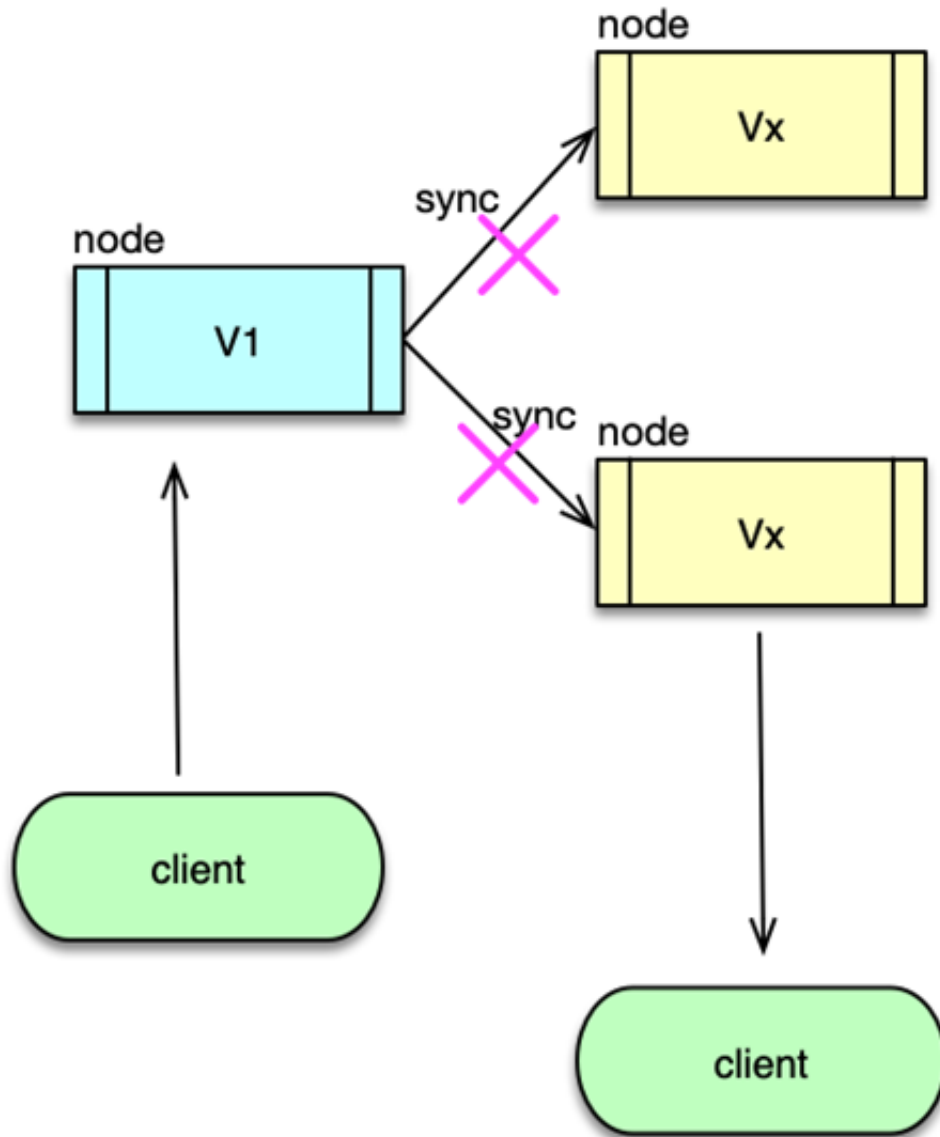
CAP 定理的第一个要素是 Consistency 一致性。一致性的英文含义是指“all nodes see the same data at the same time”。即所有节点在任意时间，被访问返回的数据完全一致。CAP 作者 Brewer 的另外一种解释是在写操作之后的读指令，必须得到的是写操作写入的值，或者写操作之后新更新的值。从服务端的视角来看，就是在 Client 写入一个更新后，Server 端如何同步这个新值到整个系统，从而保证整个系统的这个数据都相同。而从客户端的视角来看，则是并发访问时，在变更数据后，如何获取到最新值。

Availability 可用性



CAP 定理的第二个要素是 Availability 可用性。可用性的英文含义是指“Reads and writes always succeed”。即服务集群总能够对用户的请求给予响应。Brewer 的另外一个种解释是对于一个没有宕机或异常的节点，总能响应用户的请求。也就是说当用户访问一个正常工作的节点时，系统保证该节点必须给用户一个响应，可以是正确的响应，也可以是一个老的甚至错误的响应，但是不能没有响应。从服务端的视角来看，就是服务节点总能响应用户请求，不会吞噬、阻塞请求。而从客户端视角来看，发出的请求总有响应，不会出现整个服务集群无法连接、超时、无响应的情况。

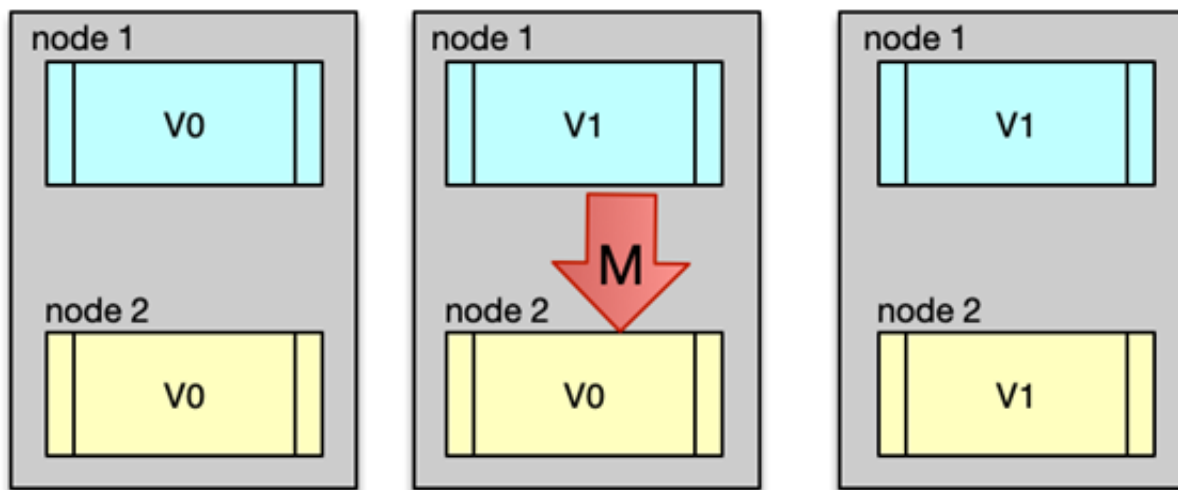
Partition Tolerance 分区容错性



第三个要素是 Partition Tolerance 分区容错性。分区容错的英文含义是指“The system continues to operate despite arbitrary message loss or failure of part of the system”。即出现分区故障或分区间通信异常时，系统仍然要对外提供服务。在分布式环境，每个服务节点都不是可靠的，不同服务节点之间的通信有可能出现问题。当某些节点出现异常，或者某些节点与其他节点之间的通信出现异常时，整个系统就产生了分区问题。从服务端的视角来看，出现节点故障、网络异常时，服务集群仍然能对外提供稳定服务，就是具有较好的分区容错性。从客户端视角来看，就是服务端的各种故障对自己透明。

正常服务场景



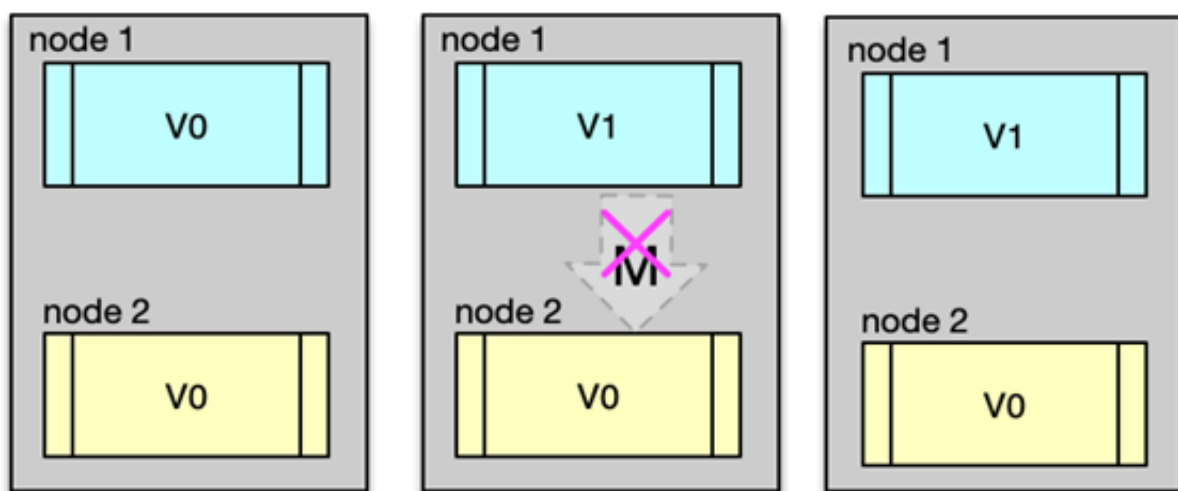


根据CAP定理，在分布式系统中这三个要素不可能三者兼顾，最多只能同时满足两点。接下来，我们用最简单的2 个服务节点场景，简要证明一下 CAP 定理。

如图所示，网络上有 2 个服务节点 Node1 和 Node2，它们之间通过网络连通组成一个分布式系统。在正常工作的业务场景，Node1 和 Node2 始终正常运行，且网络一直良好连通。

假设某初始时刻，两个节点中的数据相同，都是 V0，用户访问 Node1 和 Node2 都会立即得到 V0 的响应。当用户向 Node1 更新数据，将 V0 修改为 V1时，分布式系统会构建一个数据同步操作 M，将 V1 同步给 Node2，由于 Node1 和 Node2 都正常工作，且相互之间通信良好，Node2 中的 V0 也会被修改为 V1。此时，用户分别请求 Node1 和 Node2，得到的都是 V1，数据保持一致性，且总可以都得到响应。

网络异常场景



作为一个分布式系统，总是有多个分布的、需要网络连接的节点，节点越多、网络连接越复杂，节点故障、网络异常的情况出现的概率就会越大。要完全满足 CAP 三个元素。就意味着，如果节点之间出现了网络异常时，需要支持网络异常，即支持分区容错性，同时分布式系统还需要满足一致性和可用性。我们接下来看是否可行。

现在继续假设，初始时刻，Node1 和 Node2 的数据都是 V0，然后此时 Node1 和 Node2 之间的网络断开。用户向 Node1 发起变更请求，将 V0 变更为 V1，分布式系统准备发起同步操作 M，但由于 Node1 和 Node2 之间网络断开，同步操作 M 无法及时同步到 Node2，所以 Node2 中的数据仍然是 V0。

此时，有用户向 Node2 发起请求，由于 Node2 与 Node1 断开连接，数据没有同步，Node2 无法立即向用户返回正确的结果 V1。那怎么办呢？有两种方案。

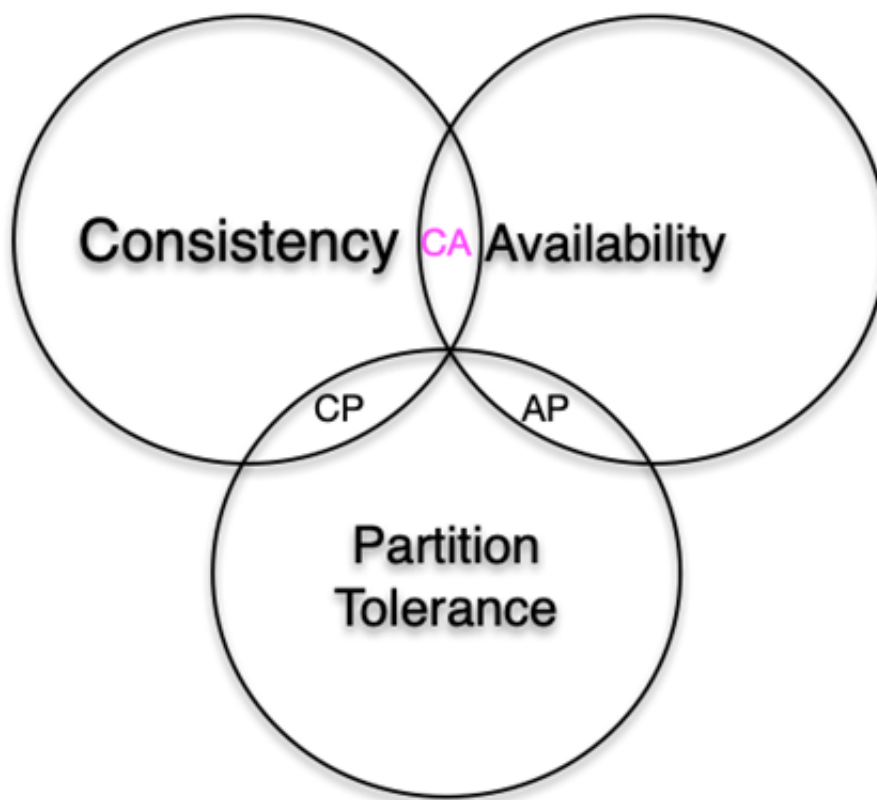
第一种方案，是牺牲一致性，Node2 向请求用户返回老数据 v0 的响应。

第二种方案，是牺牲可用性，Node2 持续阻塞请求，直到 Node1 和 Node2 之间的网络连接恢复，并且数据更新操作 M 在 Node2 上执行完毕，Node2 再给用户返回正确的 V1 操作。

至此，简要证明过程完毕。整个分析过程也就说明了，分布式系统满足分区容错性时，就无法同时满足一致性和可用性，只能二选一，也就进一步证明了分布式系统无法同时满足一致性、可用性、分区容错性这三个要素。

CAP 权衡

CA



根据 CAP 理论和前面的分析，我们知道分布式系统无法同时满足一致性、可用性、分区容错性三个要素，那我们在构建分布式系统时，应该如何选择呢？

由于这三个要素对分布式系统都非常重要，既然三个不能同时满足，那就先尽量满足两个，只舍弃其中的一个元素。

第一种方案选择是 CA，即不支持分区容错，只支持一致性和可用性。不支持分区容错性，也就意味着不允许分区异常，设备、网络永远处于理想的可用状态，从而让整个分布式系统满足一致性和可用性。

但由于分布式系统是由众多节点通过网络通信连接构建的，设备故障、网络异常是客观存在的，而且分布的节点越多，范围越广，出现故障和异常的概率也越大，因此，对于分布式系统而言，分区容错 P 是无法避免的，如果避免了 P，只能把分布式系统回退到单机单实例系统。

CP

第二种方案选择是 CP，因为分区容错 P 客观存在，即相当于放弃系统的可用性，换取一致性。那么系统在遇到分区异常时，会持续阻塞整个服务，直到分区问题解决，才恢复对外服务，这样可以保证数据的一致性。选择 CP 的业务场景比较多，特别是对数据一致性特别敏感的业务最为普遍。比如在支付交易领域，Hbase 等分布式数据库领域，都要优先保证数据的一致性，在出现网络异常时，系统就会暂停服务处理。分布式系统中，用来分发及订阅元数据的 Zookeeper，也是选择优先保证 CP 的。因为数据的一致性是一些系统的基本要求，否则，银行系统 0 余额大量取现，数据库系统访问，随机返回新老数据都会引发一系列的严重问题。

AP

第三种方案选择是 AP，由于分区容错 P 客观存在，即相当于放弃系统数据的一致性，换取可用性。这样，在系统遇到分区异常时，节点之间无法通信，数据处于不一致的状态，为了保证可用性，服务节点在收到用户请求后立即响应，那只能返回各自新老不同的数据。这种舍弃一致性，而保证系统在分区异常下的可用性，在互联网系统中非常常见。比如微博多地部署，如果不同区域的网络中断，区域内的用户仍然发微博、相互评论和点赞，但暂时无法看到其他区域用户发布的新微博和互动状态。对于微信朋友圈也是类似。还有如 12306 的火车票系统，在节假日高峰期抢票时，偶尔也会遇到，反复看到某车次有余票，但每次真正点击购买时，却提示说没有余票。这样，虽然很小一部分功能受限，但系统整体服务稳定，影响非常有限，相比 CP，用户体验会更佳。

CAP 问题及误区

CAP 理论极大的促进了分布式系统的发展，但随着分布式系统的演进，大家发现，其实 CAP 经典理论其实过于理想化，存在不少问题和误区。

首先，以互联网场景为例，大中型互联网系统，主机数量众多，而且多区域部署，每个区域有多个 IDC。节点故障、网络异常，出现分区问题很常见，要保证用户体验，理论上必须保证服务的可用性，选择 AP，暂时牺牲数据的一致性，这是最佳的选择。

但是，当分区异常发生时，如果系统设计的不够良好，并不能简单的选择可用性或者一致性。例如，当分区发生时，如果一个区域的系统必须要访问另外一个区域的依赖子服务，才可以正常提供服务，而此时网络异常，无法访问异地的依赖子服务，这样就会导致服务的不可用，无法支持可用性。同时，对于数据的一致性，由于网络异常，无法保证数据的一致性，各区域数据暂时处于不一致的状态。在网络恢复后，由于待同步的数据众多且复杂，很容易出现不一致的问题，同时某些业务操作可能跟执行顺序有关，即便全部数据在不同区域间完成同步，但由于执行顺序不同，导致最后结果也会不一致。长期多次分区异常后，会累积导致大量的数据不一致，从而持续影响用户体验。

其次，在分布式系统中，分区问题肯定会发生，但却很少发生，或者说相对于稳定工作的时间，会很短且很小概率。当不存在分区时，不应该只选择 C 或者 A，而是可以同时提供一致性和可用性。

再次，同一个系统内，不同业务，同一个业务处理的不同阶段，在分区发生时，选择一致性和可用性的策略可能都不同。比如前面讲的 12306 购票系统，车次查询功能会选择 AP，购票功能在查询阶段也选择 AP，但购票功能在支付阶段，则会选择 CP。因此，在系统架构或功能设计时，并不能简单选择 AP 或者 CP。

而且，系统实际运行中，对于 CAP 理论中的每个元素，实际并不都是非黑即白的。比如一致性，有强一致性，也有弱一致性，即便暂时大量数据不一致，在经历一段时间后，不一致数据会减少，不一致率会降低。又如可用性，系统可能会出现部分功能异常，其他功能正常，或者压力过大，只能支持部分用户的请求的情况。甚至分区也可以有一系列中间状态，区域网络完全中断的情况较少，但网络通信条件却可以在 0~100% 之间连续变化，而且系统内不同业务、不同功能、不同组件对分区还可以有不同的认知和设置。

最后，CAP 经典理论，没有考虑实际业务中网络延迟问题，延迟自始至终都存在，甚至分区异常 P 都可以看作一种延迟，而且这种延迟可以是任意时间，1 秒、1 分钟、1 小时、1 天都有可能，此时系统架构和功能设计时就要考虑，如何进行定义区分及如何应对。

这些问题，传统的 CAP 经典理论并没有给出解决方案，开发者如果简单进行三选二，就会进入误区，导致系统在运行中问题连连。

## 第31讲：如何设计足够可靠的分布式缓存体系，以满足大中型移动互联网系统的需要？

上一课时我们了解了为什么不能设计出同时满足一致性、可用性、分区容错性的分布式系统，本课时我们来具体看下，工作中应该如何设计分布式系统，以满足大中型互联网系统的需求。

### 传统 CAP 的突破

随着分布式系统的不断演进，会不断遇到各种问题，特别是当前，在大中型互联网系统的演进中，私有云、公有云并行发展且相互融合，互联网系统的部署早已突破单个区域，系统拓扑走向全国乃至全球的多区域部署。在践行传统的经典 CAP 理论的同时，需要认识到 CAP 三要素的复杂性，不能简单的对 CAP 理论进行三选二，需要根据业务特点、部署特点，对 CAP 理论进行创新、修正及突破。

甚至 CAP 理论的提出者 Eric Brewer 自己也在 CAP 理论提出的 12 年后，即在 2012 年，对 CAP 理论，特别是 CAP 使用中的一些误区，进一步进行修正、拓展及演进说明。Brewer 指出，CAP 理论中经典的三选二公式存在误导性，CAP 理论的经典实践存在过于简化三种要素，以及三要素之间的相互关系的问题。他同时把 CAP 与 ACID、BASE 进行比较，分析了 CAP 与延迟的关系，最后还重点分析了分布式系统如何应对分区异常的问题。

要突破经典的 CAP 理论和实践，要认识到 CAP 三要素都不是非黑即白，而是存在一系列的可能性，要在实际业务场景中对分布式系统，进行良好的架构设计，这是一个很大的挑战。

在系统实际运行过程中，大部分时间，分区异常不会发生，此时可以提供良好的一致性和可用性。同时，我们需要在系统架构设计中，在分析如何实现业务功能、系统 SLA 指标实现等之外，还要考虑整个系统架构中，各个业务、模块、功能、系统部署如何处理潜在的分区问题。

要良好处理潜在的分区问题，可以采用如下步骤。

首先，要考虑如何感知分区的发生，可以通过主动探测、状态汇报、特殊时间/特殊事件预警、历史数据预测等方式及时发现分区。

其次，如果发现分区，如何在分区模式下进行业务处理。可以采用内存缓冲、队列服务保存数据后，继续服务，也可以对敏感功能直接停止服务，还可以对分区进行进一步细分，如果是短时间延迟，可以部分功能或请求阻塞等待结果，其他功能和请求快速返回本地老数据；如果分区时长超过一定阈值，进行部分功能下线，只提供部分核心功能。

最后，在分区异常恢复后，如何同步及修复数据，建立补偿机制应对分区模式期间的错误。如系统设计中引入消息队列，在分区模式期间，变更的数据用消息队列进行保存，分区恢复后，消息处理机从消息队列中进行数据读取及修复。也可以设计为同步机制，分区异常时，记录最后同步的位置点，分区恢复后，从记录的位置点继续同步数据。还可以在分区时，分布式系统的各区记录自己没有同步出去的数据，然后在分区恢复后，主动进行异地数据比较及合并。最后，还可以在故障恢复后通过数据扫描，对比分区数据，进行比较及修复。

## BASE 理论

BASE 理论最初由 Brewer 及他的同事们提出。虽然比较久远，但在当前的互联网界活力更盛。各大互联网企业，在构建大中型规模的分布式互联网系统，包括各种基于私有云、公有云及多云结合的分布式系统时，在尽力借鉴 CAP 理论与实践的同时，还充分验证和实践了 BASE 理论，并将其作为 CAP 理论的一种延伸，很好的应用在互联网各种系统中。

BASE 理论及实践是分布式系统对一致性和可用性权衡后的结果。其基本思想是分布式系统各个功能要适当权衡，尽力保持整个系统稳定可用，即便在出现局部故障和异常时，也确保系统的主体功能可用，确保系统的最终一致性。

BASE 理论也包括三要素，即 Basically Availabe 基本可用、Soft state 软状态和 Eventual Consistency 最终一致性。

Basically Availabe 基本可用

基本可用是指分布式系统在出现故障时，允许损失部分可用性。比如可以损失部分 SLA，如响应时间适当增加、处理性能适当下降，也可以损失部分周边功能、甚至部分核心功能。最终保证系统的主体基本稳定，核心功能基本可用的状态。如淘宝、京东在双十一峰值期间，请求会出现变慢，但少许延迟后，仍然会返回正确结果，同时还会将部分请求导流到降级页面等。又如微博在突发故障时，会下线部分周边功能，将资源集中用于保障首页 feed 刷新、发博等核心功能。

Soft state 软状态

软状态是指允许系统存在中间状态。故障发生时，各分区之间的数据同步出现延时或暂停，各区域的数据处于不一致的状态，这种状态的出现，并不影响系统继续对外提供服务。这种节点不一致的状态和现象就是软状态。

Eventual Consistency 最终一致性

最终一致性，是指分布式系统不需要实时保持强一致状态，在系统故障发生时，可以容忍数据的不一致，在系统故障恢复后，数据进行同步，最终再次达到一致的状态。

BASE 理论是面向大中型分布式系统提出的，它更适合当前的大中型互联网分布式系统。

首先用户体验第一，系统设计时要优先考虑可用性。

其次，在故障发生时，可以牺牲部分功能的可用性，牺牲数据的强一致性，来保持系统核心功能的可用性。

最后，在系统故障恢复后，通过各种策略，确保系统最终再次达到一致。

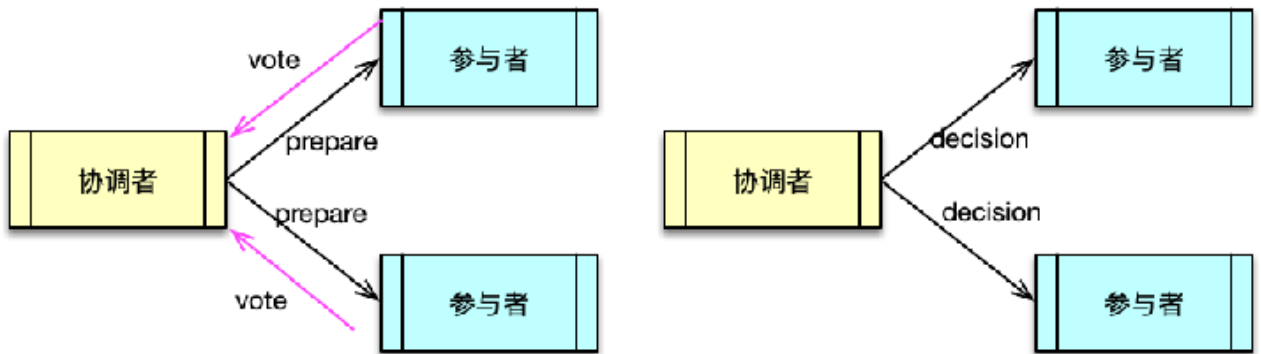
一致性问题及应对

分布式系统中，为了保持系统的可用性和性能，系统中的数据需要存储多个副本，这些副本分布在不同的物理机上，如果服务器、网络出现故障，就会导致部分数据副本写入成功，部分数据副本写入失败，这就会导致各个副本之间数据不一致，数据内容冲突，也就造成了数据的不一致。因此，为了保持分布式系统的一致性，核心就是如何解决分布式系统中的数据一致性。

保持数据一致性的方案比较多，比较常见的方案有，分布式事务，主从复制，业务层消息总线等。

分布式事务

分布式事务在各节点均能正常执行事务内一系列操作才会提交，否则就进行回滚，可以保持系统内数据的强一致。分布式事务应用比较广泛，比如跨行转账，用户甲向用户乙转账，甲账户需要减少，乙账户需要增加对应金额，这两个操作就必须构成一个分布式事务。还有其他场景，比如 12306 中支付出票、支付宝买入基金等，都需要保持对应操作的事务性。



分布式事务的具体方案较多，典型有 2PC 两阶段提交、3PC 三阶段提交、Paxos、Zab、Raft 等。

两阶段提交方案中，系统包括两类节点，一类是协调者，一类是事务参与者。协调者一般只有一个，参与者可以理解为数据副本的数量，一般有多个。

两阶段提交的执行分为请求阶段和提交阶段两部分。在请求阶段，协调者将通知事务参与者准备提交或取消事务，通知完毕后，事务参与者就开始进行表决。在表决中，参与者如果本地作业执行成功，则表决同意，如果执行失败，则表决取消，然后把表决回复给协调者。然后进入提交阶段。

在提交阶段，协调者将基于第一阶段的表决结果进行决策是提交事务还是取消事务。决策方式是所有参与者表决同意则决策提交，否则决策取消。然后协调者把决策结果分发给所有事务参与者。事务参与者接受到协调者的决策后，执行对应的操作。

三阶段提交与两阶段提交类似，只是在协调者、参与者都引入了超时机制，而且把两阶段提交中的第一阶段分拆成了 2 步，即先询问再锁资源。

分布式事务中 Paxos、Zab、Raft 等方案的基本思想类似。在每个数据副本附带版本信息，每次写操作保证写入大于  $N/2$  个节点，同时每次读操作也保证从大于  $N/2$  个节点读，以多数派作为最终决策。这种仲裁方式在业界使用比较广泛，比如亚马逊的 Dynamo 存储也是类似，Dynamo 的决策更简洁，只要写操作数 + 读操作数大于节点数即可。一般整个仲裁过程由协调者进行，当然也可以像 Dynamo 那样，支持由业务 Client 决策也没问题，更有弹性，因为可以由业务按各种策略选择。在仲裁后，仲裁者可以选择正确的版本数据，甚至在某些场景下可以将不同版本的数据合并成一个新数据。

## 主从复制

主从复制也是一种使用较为广泛的一致性方案。在 Mysql 等各种 DB 中广泛使用，之前课程中讲到的 Redis 也是采用主从复制来保持主从数据一致的。

除了从数据层保证一致性，还可以在上层业务层，通过消息总线分发，来更新缓存及存储体系，这也是互联网企业在进行异地多活方案设计时经常会考虑到的方案。

消息总线在各区域相互分发消息，有 push 推和 pull 拉两种方案。一般来讲，pull 拉的方式，由于拉取及拉取后的执行过程对分发是可以感知，在网络异常时，更容易保障数据的一致性。

## 分布式系统多区数据一致性案例

如图所示，是微博进行多区数据一致性保障案例。消息是通过消息中间件 wmb 进行分发的。wmb 两边分别为分布式系统的 2 个区域。每个区域所有的用户写操作，都会封装成一条消息，业务消息会首先写入消息队列服务，然后消息队列处理机读取消息队列，并进行缓存和 DB 的更新。在业务消息写入消息队列服务时，wmb 会同时将这条消息分发给其他所有异地区子系统。分发的方式是，wmb 本地组件先将消息写入本地队列，然后 wmb 异地组件 Client 再读取。当分区故障发生时，异地读取失败，消息仍然在各区的消息队列中，不会丢失。分区故障过程中，系统的各区子系统只处理本地事件。在分区故障排除后，wmb Client 继续读取异地消息，然后由消息处理机执行，最终实现数据的一致性。

由于 wmb 通过消息队列机方式从业务层面进行同步，分区故障发生时，各区都是先执行本地，分区恢复后再执行异地，所有事件在各区的执行顺序可能会有差异，在某些极端场景下，可能会导致数据不一致。所以，微博只用 wmb 来更新缓存，DB 层仍然采用主从复制的方式进行强一致保障。这样即便故障恢复期间，可能存在少量缓存数据暂时不一致，由于恢复数据时采用了更短的过期时间，这部分数据在从 DB 重新加载后，仍然能保持数据的最终一致性。同时，微博不用 DB 数据更新缓存，是由于缓存数据结构过于复杂，而且经常需要根据业务需要进行扩展，一条缓存记录会涉及众多 DB，以及 Redis 中多项纪录，通过 DB 同步数据触发更新缓存涉及因素太多，不可控。所以微博在尝试 DB 驱动缓存更新方案失败后，就改为 wmb 消息队列方式进行缓存更新。



## 第32讲：一个典型的分布式缓存系统是什么样的？

本课时我们具体看下一个典型的分布式缓存系统是什么样的。

### 分布式 Redis 服务

由于本课程聚焦于缓存，接下来，我将以微博内的 分布式 Redis 服务系统为例，介绍一个典型的分布式缓存系统的组成。

微博的 Redis 服务内部也称为 RedisService。RedisService 的整体架构如图所示。主要分为Proxy、存储、集群管理、配置中心、Graphite，5 个部分。

RedisService 中的 Proxy 是无状态多租户模型，每个 Proxy 下可以挂载不同的业务存储，通过端口进行业务区分。

存储基于 Redis 开发，但在集群数据存储时，只保留了基本的存储功能，支持定制的迁移功能，但存储内部无状态，不存储 key-slot 映射关系。

配置中心用于记录及分发各种元数据，如存储 Proxy 的 IP、端口、配置等，在发生变化时，订阅者可以及时感知。

Graphite 系统用于记录并展现系统、业务，组件以及实例等的状态数据。

ClusterManager 用于日常运维管理，业务 SLA 监控，报警等。同时 ClusterManager 会整合 Proxy、Redis 后端存储以及配置中心，对业务数据进行集群管理

### 多租户 Proxy

RedisService 中的 Proxy 无任何状态，所有 Proxy 实例的启动参数相同。但 Proxy 启动前，clusterManager 会在配置中心设置该实例的业务及存储配置信息，Proxy 启动后，到配置中心通过自己的 IP 来获取并订阅配置，然后进行初始化。Proxy 与后端 Redis 存储采用长连接，当 Client 并发发送请求到 Proxy 后，Proxy 会将请求进行打包，并发地以 pipeline 的方式批量访问后端，以提升请求效率。对于多租户 Proxy，由于不同业务的存储位置可能不同，因此对每个请求需要进行业务区分，一般有 2 种方式进行区分。

方案 1，按照 key 的 namespace 前缀进行业务区分，比如 Client 分别请求 user、graph、feed 业务下的 key k1，业务 Client 分别构建 {user}k1、{graph}k1、{feed}k1，然后发送给 Proxy，Proxy 解析 key 前缀确定 key 对应的业务。

方案 2，对每个业务分配一个业务端口，不同业务访问自己的端口，Proxy 会根据端口确定业务类型。这种类型不需要解析 key 前缀，不需要重构请求，性能更为高效。但需要为业务配置端口，增加管理成本，实践上，由于业务 Redis 资源一般会采用不同端口，所以业务 Proxy 可以采用业务资源分片的最小端口来作为业务端口标志。

### Redis 数据存储

RedisService 中的 Redis 存储基于 Redis 5.0 扩展，内部称 wredis，wredis 不存储 key-slot 映射，只记录当前实例中存储的 slot 的 key 计数。wredis 处理任何收到的操作命令，而数据分片访问的正确性由访问端确保。在每日低峰时段，clusterManager 对 Redis 存储进行扫描，发现 slot 存储是否存在异常。因为微博中有大量的小 value key，如果集群中增加 key-slot 映射，会大大增大存储成本，通过消除 key-slot 映射等相关优化，部分业务可以减少 20% 以上的存储容量。

wredis 支持 slot 的同步迁移及异步迁移。同时支持热升级，可以毫秒级完成组件升级。wredis 也支持全增量复制，支持微博内部扩展的多种数据结构。热升级、全增量复制、数据结构扩展等，在之前的课时中有介绍，具体可以参考之前讲的“Redis 功能扩展”课时的内容。

## 配置中心 configService

微博的配置中心，内部称为 configService，是微博内部配置元数据管理的基础组件。configService 自身也是多 IDC 部署的，配置信息通过多版本数据结构存储，支持版本回溯。同时配置数据可以通过 merkle hash 树进行快速一致性验证。RedisService 中的所有业务、资源、Proxy 的配置都存储在 configService 中，由 cluster 写入并变更，Proxy、业务 Client 获取并订阅所需的配置数据。configService 在配置节点发生变更时，会只对节点进行事件通知，订阅者无需获取全量数据，可以大大减轻配置变更后的获取开销。

ClusterManager 是一个运维后台。主要用于运维工作，如后端资源、Proxy 的实例部署，配置变更，版本升级等。也用于数据的集群管理，clusterManager 内部会存储业务数据的集群映射，并在必要时进行数据迁移和故障转移。迁移采用 slot 方式，可以根据负载进行迁移流量控制，同时会探测集群内的节点状态，如在 wredis 的 master 异常后，从 slave 中选择一个新的 master，并重建主从关系。clusterManager 还支持业务访问的 Proxy 域名管理，监控集群节点的实例状态，监控业务的 SLA 指标，对异常进行报警，以便运维及时进行处理。

## 集群数据同步

RedisService 中的数据存储在多个区域，每个区域都有多个 IDC。部署方式是核心内网加公有云的方式。使用公有云，主要是由微博的业务特点决定的，在突发事件或热点事件发生时，很容易形成流量洪峰，读写 TPS 大幅增加，利用公有云可以快速、低成本的扩展系统，大幅增加系统处理能力。根据业务特点，wredis 被分为缓存和存储类型。对于 Redis 缓存主要通过消息总线进行驱动更新，而对于 Redis 存储则采用主从复制更新。更新方式不同，主要是因为 Redis 作为缓存类型的业务数据，在不同区或者不同 IDC 的热点数据不同，如果采用主从复制，部署从库的 IDC，会出现热数据无法进入缓存，同时冷数据无法淘汰的问题，因为从库的淘汰也要依赖主库进行。而对于 Redis 作存储的业务场景，由于缓存存放全量数据，直接采用主从复制进行数据一致性保障，这样最便捷。

# 第十一章：应用场景案例解析

---

## 第33讲：如何为秒杀系统设计缓存体系？

本课时我们具体讲解如何为秒杀系统设计缓存体系。

### 秒杀系统分析

互联网电商为了吸引人气，经常会对一些商品进行低价秒杀售卖活动。比如几年前小米的不定期新品发售，又如当前每年定期举行双11、双12中的特价商品售卖。秒杀售卖时，大量消费者蜂拥而至，给电商带来了极大的人气，也给电商背后的服务系统带来了超高的并发访问负荷。

在不同电商、不同的秒杀活动，秒杀系统售卖的商品、销售策略大不相同，但秒杀背后的秒杀系统却有很大的相似性，基本都有以下这些共同特点。

首先，秒杀业务简单，每个秒杀活动售卖的商品是事先定义好的，这些商品有明确的类型和数量，卖完即止。

其次，秒杀活动定时上架，而且会提供一个秒杀入口，消费者可以在活动开始后，通过这个入口进行抢购秒杀活动。

再次，秒杀活动由于商品售价低廉，广泛宣传，购买者远大于商品数，开始售卖后，会被快速抢购一空。

最后，由于秒杀活动的参与者众多，远超日常访客数量，大量消费者涌入秒杀系统，还不停的刷新访问，短时间内给系统带来超高的并发流量，直到活动结束，流量消失。

分析了秒杀系统的特点，很容易发现，秒杀系统实际就是一个有计划的低价售卖活动，活动期间会带来N倍爆发性增长的瞬时流量，活动后，流量会快速消失。因此，秒杀活动会给后端服务带来如下的技术挑战。

首先，秒杀活动持续时间短，但访问冲击量大，秒杀系统需要能够应对这种爆发性的类似攻击的访问模型。

其次，业务的请求量远远大于售卖量，大部分是最终无法购买成功的请求，秒杀系统需要提前规划好处理策略；

而且，由于业务前端访问量巨大，系统对后端数据的访问量也会短时间爆增，需要对数据存储资源进行良好设计。

另外，秒杀活动虽然持续时间短，但活动期间会给整个业务系统带来超大负荷，业务系统需要制定各种策略，避免系统过载而宕机。

最后，由于售卖活动商品价格低廉，存在套利空间，各种非法作弊手段层出，需要提前规划预防策略。

## 秒杀系统设计

在设计秒杀系统时，有两个设计原则。

首先，要尽力将请求拦截在系统上游，层层设阻拦截，过滤掉无效或超量的请求。因为访问量远远大于商品数量，所有的请求打到后端服务的最后一步，其实并没有必要，反而会严重拖慢真正能成交的请求，降低用户体验。

其次，要充分利用缓存，提升系统的性能和可用性。

秒杀系统专为秒杀活动服务，售卖商品确定，因此可以在设计秒杀商品页面时，将商品信息提前设计为静态信息，将静态的商品信息以及常规的CSS、JS、宣传图片等静态资源，一起独立存放到CDN节点，加速访问，且降低系统访问压力。

在访问前端也可以制定种种限制策略，比如活动没开始时，抢购按钮置灰，避免抢先访问，用户抢购一次后，也将按钮置灰，让用户排队等待，避免反复刷新。

用户所有的请求进入秒杀系统前，通过负载均衡策略均匀分发到不同 Web 服务器，避免节点过载。在 Web 服务器中，首先进行各种服务预处理，检查用户的访问权限，识别并发刷订单的行为。同时在真正服务前，也要进行服务前置检查，避免超售发生。如果发现售出数量已经达到秒杀数量，则直接返回结束。

秒杀系统在处理抢购业务逻辑时，除了对用户进行权限校验，还需要访问商品服务，对库存进行修改，访问订单服务进行订单创建，最后再进行支付、物流等后续服务。这些依赖服务，可以专门为秒杀业务设计排队策略，或者额外部署实例，对秒杀系统进行专门服务，避免影响其他常规业务系统。

在秒杀系统设计中，最重要的是在系统开发之初就进行有效分拆。首先分拆秒杀活动页面的内容，将静态内容分拆到 CDN，动态内容才通过接口访问。其次，要将秒杀业务系统和其他业务系统进行功能分拆，尽量将秒杀系统及依赖服务独立分拆部署，避免影响其他核心业务系统。

由于秒杀的参与者远大于商品数，为了提高抢购的概率，时常会出现一些利用脚本和僵尸账户并发频繁调用接口进行强刷的行为，秒杀系统需要构建访问记录缓存，记录访问 IP、用户的访问行为，发现异常访问，提前进行阻断及返回。同时还需要构建用户缓存，并针对历史数据分析，提前缓存僵尸强刷专业户，方便在秒杀期间对其进行策略限制。这些访问记录、用户数据，通过缓存进行存储，可以加速访问，另外，对用户数据还进行缓存预热，避免活动期间大量穿透。

在业务请求处理时，所有操作尽可能由缓存交互完成。由于秒杀商品较少，相关信息全部加载到内存，把缓存暂时当作存储用，并不会带来过大成本负担。

为秒杀商品构建商品信息缓存，并对全部目标商品进行预热加载。同时对秒杀商品构建独立的库存缓存，加速库存检测。这样通过秒杀商品列表缓存，进行快速商品信息查询，通过库存缓存，可以快速确定秒杀活动进程，方便高效成交或无可售商品后的快速检测及返回。在用户抢购到商品后，要进行库存事务变更，进行库存、订单、支付等相关的构建和修改，这些操作可以尽量由系统只与缓存组件交互完成初步处理。后续落地等操作，必须要入 DB 库的操作，可以先利用消息队列，记录成交事件信息，然后再逐步分批执行，避免对 DB 造成过大压力。

总之，在秒杀系统中，除了常规的分拆访问内容和服务，最重要的是尽量将所有数据访问进行缓存化，尽量减少 DB 的访问，在大幅提升系统性能的同时，提升用户体验。

## 第34讲：如何为海量计数场景设计缓存体系？

在上一课时我们讲解了如何为秒杀系统进行缓存设计，在本课时我们将具体讲解如何为海量计数场景设计缓存服务。

### 计数常规方案

计数服务在互联网系统中非常常见，用户的关注粉丝数、帖子数、评论数等都需要进行计数存储。计数的存储格式也很简单，key 一般是用户 uid 或者帖子 id 加上后缀，value 一般是 8 字节的 long 型整数。

最常见的计数方案是采用缓存 + DB 的存储方案。当计数变更时，先变更计数 DB，计数加 1，然后再变更计数缓存，修改计数存储的 Memcached 或 Redis。这种方案比较通用且成熟，但在高并发访问场景，支持不够友好。在互联网社交系统中，有些业务的计数变更特别频繁，比如微博 feed 的阅读数，计数的变更次数和访问次数相当，每秒十万到百万级以上的更新量，如果用 DB 存储，会给 DB 带来巨大的压力，DB 就会成为整个计数服务的瓶颈所在。即便采用聚合延迟更新 DB 的方案，由于总量特别大，同时请求均衡分散在大量不同的业务端，巨大的写压力仍然是 DB 的不可承受之重。因此这种方案只适合中小规模的计数服务使用。

在 Redis 问世并越来越成熟后，很多互联网系统会直接把计数全部存储在 Redis 中。通过 hash 分拆的方式，可以大幅提升计数服务在 Redis 集群的写性能，通过主从复制，在 master 后挂载多个从库，利用读写分离，可以大幅提升计数服务在 Redis 集群的读性能。而且 Redis 有持久化机制，不会丢数据，在很多大中型互联网场景，这都是一个比较适合的计数服务方案。

在互联网移动社交领域，由于用户基数巨大，每日发表大量状态数据，且相互之间有大量的交互动作，从而产生了海量计数和超高并发访问，如果直接用 Redis 进行存储，会带来巨大的成本和性能问题。

### 海量计数场景

以微博为例，系统内有大量的待计数对象。如从用户维度，日活跃用户 2 亿+，月活跃用户接近 5 亿。从 Feed 维度，微博历史 Feed 有数千亿条，而且每日新增数亿条的新 Feed。这些用户和 Feed 不但需要进行计数，而且需要进行多个计数。比如，用户维度，每个用户需要记录关注数、粉丝数、发表 Feed 数等。而从 Feed 维度，每条 Feed 需要记录转发数、评论数、赞、阅读等计数。

而且，在微博业务场景下，每次请求都会请求多个对象的多个计数。比如查看用户时，除了获取该用户的基本信息，还需要同时获取用户的关注数、粉丝数、发表 Feed 数。获取微博列表时，除了获取 Feed 内容，还需要同时获取 Feed 的转发数、评论数、赞数，以及阅读数。因此，微博计数服务的总访问量特别大，很容易达到百万级以上的 QPS。

因此，在海量计数高并发访问场景，如果采用缓存 + DB 的架构，首先 DB 在计数更新就会存在瓶颈，其次，单个请求一次请求数十个计数，一旦缓存 miss，穿透到 DB，DB 的读也会成为瓶颈。因为 DB 能支撑的 TPS 不过 3000~6000 之间，远远无法满足高并发计数访问场景的需要。

采用 Redis 全量存储方案，通过分片和主从复制，读写性能不会成为主要问题，但容量成本却会带来巨大开销。

因为，一方面 Redis 作为通用型存储来存储计数，内存存储效率低。以存储一个 key 为 long 型 id、value 为 4 字节的计数为例，Redis 至少需要 65 个字节左右，不同版本略有差异。但这个计数理论只需要占用 12 个字节即可。内存有效负荷只有  $12/65=18.5\%$ 。如果再考虑一个 long 型 id 需要存 4 个不同类型的 4 字节计数，内存有效负荷只有  $(8+16)/(65*4)=9.2\%$ 。

另一方面，Redis 所有数据均存在内存，单存储历史千亿级记录，单份数据拷贝需要 10T 以上，要考虑核心业务上 1 主 3 从，需要 40T 以上的内存，再考虑多 IDC 部署，轻松占用上百 T 内存。就按单机 100G 内存来算，计数服务就要占用上千台大内存服务器。存储成本太高。

### 海量计数服务架构

为了解决海量计数的存储及访问的问题，微博基于 Redis 定制开发了计数服务系统，该计数服务兼容 Redis 协议，将所有数据分别存储在内存和磁盘 2 个区域。首先，内存会预分配 N 块大小相同的 Table 空间，线上一般每个 Table 占用 1G 字节，最大分配 10 个左右的 Table 空间。首先使用 Table0，当存储填充率超过阈值，就使用 Table1，依次类推。每个 Table 中，key 是微博 id，value 是自定义的多个计数。

微博的 id 按时间递增，因此每个内存 Table 只用存储一定范围内的 id 即可。内存 Table 预先按设置分配为相同 size 大小的 key-value 槽空间。每插入一个新 key，就占用一个槽空间，当槽位填充率超过阈值，就滚动使用下一个 Table，当所有预分配的 Table 使用完毕，还可以根据配置，继续从内存分配更多新的 Table 空间。当内存占用达到阈值，就会把内存中 id 范围最小的 Table 落盘到 SSD 磁盘。落盘的 Table 文件称为 DDB。每个内存 Table 对应落盘为 1 个 DDB 文件。

计数服务会将落盘 DDB 文件的索引记录在内存，这样当查询需要从内存穿透到磁盘时，可以直接定位到磁盘文件，加快查询速度。

计数服务可以设置 Schema 策略，使一个 key 的 value 对应存储多个计数。每个计数占用空间根据 Schema 确定，可以精确到 bit。key 中的各个计数，设置了最大存储空间，所以只能支持有限范围内的计数。如果计数超过设置的阈值，则需要将这个 key 从 Table 中删除，转储到 aux dict 辅助词典中。

同时每个 Table 负责一定范围的 id，由于微博 id 随时间增长，而非逐一递增，Table 滚动是按照填充率达到阈值来进行的。当系统发生异常时，或者不同区域网络长时间断开重连后，在老数据修复期间，可能在之前的 Table 中插入较多的计数 key。如果旧 Table 插入数据量过大，超过容量限制，或者持续搜索存储位置而不得，查找次数超过阈值，则将新 key 插入到 extend dict 扩展词典中。

微博中的 feed 一般具有明显的冷热区分，并且越新的 feed 越热，访问量越大，越久远的 feed 越冷。新的热 key 存放内存 Table，老的冷 key 随所在的 Table 被置换到 DDB 文件。当查询 DDB 文件中的冷 key 时，会采用多线程异步并行查询，基本不影响业务的正常访问。同时，这些冷 key 从 DDB 中查询后，会被存放到 LRU 中，从而方便后续的再次访问。

计数服务的内存数据快照仍然采用前面讲的 RDB + 滚动 AOF 策略。RDB 记录构建时刻对应的 AOF 文件 id 及 pos 位置。全量复制时，master 会将磁盘中的 DDB 文件，以及内存数据快照对应的 RDB 和 AOF 全部传送给 slave。

在之后的所有复制就是全增量复制，slave 在断开连接，再次重连 master 时，汇报自己同步的 AOF 文件 id 及位置，master 将对应文件位置之后的内容全部发送给 slave，即可完成同步。

计数服务中的内存 Table 是一个一维开放数据，每个 key-value 按照 Schema 策略占用相同的内存。每个 key-value 内部，key 和多个计数紧凑部署。首先 8 字节放置 long 型 key，然后按 Schema 设置依次存放各个计数。

key 在插入及查询时，流程如下。

首先根据所有 Table 的 id 范围，确定 key 所在的内存 Table。

然后再根据 double-hash 算法计算 hash，用 2 个 hash 函数分别计算出 2 个 hash 值，采用公示  $h1 + N * h2$  来定位查找。

在对计数插入或变更时，如果查询位置为空，则立即作为新值插入 key/value，否则对比 key，如果 key 相同，则进行计数增减；如果 key 不同，则将 N 加 1，然后进入到下一个位置，继续进行前面的判断。如果查询的位置一直不为空，且 key 不同，则最多查询设置的阈值次数，如果仍然没查到，则不再进行查询。将该 key 记录到 extend dict 扩展词典中。

在对计数 key 查找时，如果查询的位置为空，说明 key 不存在，立即停止。如果 key 相同，返回计数，否则 N 加 1，继续向后查询，如果查询达到阈值次数，没有遇到空，且 key 不同，再查询 aux dict 辅助字典和 extend dict 扩展字典，如果也没找到该 key，则说明该 key 不存在，即计数为 0。

海量计数服务收益

微博计数服务，多个计数按 Schema 进行紧凑存储，共享同一个 key，每个计数的 size 按 bit 设计大小，没有额外的指针开销，内存占用只有 Redis 的 10% 以下。同时，由于 key 的计数 size 固定，如果计数超过阈值，则独立存储 aux dict 辅助字典中。

同时由于一个 key 存储多个计数，同时这些计数一般都需要返回，这样一次查询即可同时获取多个计数，查询性能相比每个计数独立存储的方式提升 3~5 倍。

## 第35讲：如何为社交feed场景设计缓存体系？

在上一课时我们讲解了如何为海量计数场景进行缓存设计，本课时中我将讲解如何为社交 Feed 场景设计缓存体系。

### Feed 流场景分析

Feed 流是很多移动互联网系统的重要一环，如微博、微信朋友圈、QQ 好友动态、头条/抖音信息流等。虽然这些产品形态各不相同，但业务处理逻辑却大体相同。用户日常的“刷刷刷”，就是在获取 Feed 流，这也是 Feed 流的一个最重要应用场景。用户刷新获取 Feed 流的过程，对于服务后端，就是一个获取用户感兴趣的 Feed，并对 Feed 进行过滤、动态组装的过程。

接下来，我将以微博为例，介绍用户在发出刷新 Feed 流的请求后，服务后端是如何进行处理的。

获取 Feed 流操作是一个重操作，后端数据处理存在 100 ~ 1000 倍以上的读放大。也就是说，前端用户发出一个接口请求，服务后端需要请求数百甚至数千条数据，然后进行组装处理并返回响应。因此，为了提升处理性能、快速响应用户，微博 Feed 平台重度依赖缓存，几乎所有的数据都从缓存获取。如用户的关注关系从 Redis 缓存中获取，用户发出的 Feed 或收到特殊 Feed 从 Memcached 中获取，用户及 Feed 的各种计数从计数服务中获取。

### Feed 流流程分析

Feed 流业务作为微博系统的核心业务，为了保障用户体验，SLA 要求较高，核心接口的可用性要达到 4 个 9，接口耗时要在 50~100ms 以内，后端数据请求平均耗时要在 3~5ms 以内，因此为了满足亿级庞大用户群的海量并发访问需求，需要对缓存体系进行良好架构且不断改进。

在 Feed 流业务中，核心业务数据的缓存命中率基本都在 99% 以上，这些缓存数据，由 Feed 系统进行多线程并发获取及组装，从而及时发送响应给用户。

Feed 流获取的处理流程如下。

首先，根据用户信息，获取用户的关注关系，一般会得到 300~2000 个关注用户的 UID。

然后，再获取用户自己的 Feed inbox 收件箱。收件箱主要存放其他用户发表的供部分特定用户可见的微博 ID 列表。

接下来，再获取所有关注列表用户的微博 ID 列表，即关注者发表的所有用户或者大部分用户可见的 Feed ID 列表。这些 Feed ID 列表都以 vector 数组的形式存储在缓存。由于一般用户的关注数会达到数百甚至数千，因此这一步需要获取数百或数千个 Feed vector。

然后，Feed 系统将 inbox 和关注用户的所有 Feed vector 进行合并，并排序、分页，即得到目标 Feed 的 ID 列表。



接下来，再根据 Feed ID 列表获取对应的 Feed 内容，如微博的文字、视频、发表时间、源微博 ID 等。

然后，再进一步获取所有微博的发表者 user 详细信息、源微博内容等信息，并进行内容组装。

之后，如果用户设置的过滤词，还要将这些 Feed 进行过滤筛选，剔除用户不感兴趣的 Feed。

接下来，再获取用户对这些 Feed 的收藏、赞等状态，并设置到对应微博中。

最后，获取这些 Feed 的转发数、评论数、赞数等，并进行计数组装。至此，Feed 流获取处理完毕，Feed 列表以 JSON 形式返回给前端，用户刷新微博首页成功完成。

## Feed 流缓存架构

Feed 流处理中，缓存核心业务数据主要分为 6 大类。

第一类是用户的 inbox 收件箱，在用户发表仅供少量用户可见的 Feed 时，为了提升访问效率，这些 Feed ID 并不会进入公共可见的 outbox 发件箱，而会直接推送到目标客户的收件箱。

第二类是用户的 outbox 发件箱。用户发表的普通微博都进入 outbox，这些微博几乎所有人都可见，由粉丝在刷新 Feed 列表首页时，系统直接拉取组装。

第三类是 Social Graph 即用户的关注关系，如各种关注列表、粉丝列表。

第四类是 Feed Content 即 Feed 的内容，包括 Feed 的文字、视频、发表时间、源微博 ID 等。

第五类是 Existence 存在性判断缓存，用来判断用户是否阅读了某条 Feed，是否赞了某条 Feed 等。对于存在性判断，微博是采用自研的 phantom 系统，通过 bloomfilter 算法进行存储的。

第六类是 Counter 计数服务，用来存储诸如关注数、粉丝数，Feed 的转发、评论、赞、阅读等各种计数。

对于 Feed 的 inbox 收件箱、outbox 发件箱，Feed 系统通过 Memcached 进行缓存，以 feed id 的一维数组格式进行存储。

对于关注列表，Feed 系统采用 Redis 进行缓存，存储格式为 longset。longset 在之前的课时介绍过，是微博扩展的一种数据结构，它是一个采用 double-hash 寻址的一维数组。当缓存 miss 后，业务 client 可以从 DB 加载，并直接构建 longset 的二进制格式数据作为 value 写入 Redis，Redis 收到后直接 restore 到内存，而不用逐条加入。这样，即使用户有成千上万个关注，也不会引发阻塞。

Feed content 即 Feed 内容，采用 Memcached 存储。由于 Feed 内容有众多的属性，且时常需要根据业务需要进行扩展，Feed 系统采用 Google 的 protocol buffers 的格式进行存放。protocol buffers 序列化后的所生成的二进制消息非常紧凑，二进制存储空间比 XML 小 3~10 倍，而序列化及反序列化的性能却高 10 倍以上，而且扩展及变更字段也很方便。微博的 Feed content 最初采用 XML 和 JSON 存储，在 2011 年之后逐渐全部改为 protocol buffers 存储。

对于存在性判断，微博 Feed 系统采用自研的 phantom 进行存储。数据存储采用 bloom filter 存储结构。实际上 phantom 本身就是一个分段存储的 bloomfilter 结构。bloomFilter 采用 bit 数组来表示一个集合，整个数组最初所有 bit 位都是 0，插入 key 时，采用 k 个相互独立的 hash 函数计算，将对应 hash 位置置 1。而检测某个 key 是否存在时，通过对 key 进行多次 hash，检查对应 hash 位置是否为 1 即可，如果有一个为 0，则可以确定该 key 肯定不存在，但如果全部为 1，大概率说明该 key 存在，但该 key 也有可能不存在，即存在一定的误判率，不过这个误判率很低，一般平均每条记录占用 1.2 字节时，误判率即可降低到 1%，1.8 字节，误判率可以降到千分之一。基本可以满足大多数业务场景的需要。

对于计数服务，微博就是用前面讲到的 CounterService。CounterService 采用 schema 策略，支持一个 key 对应多个计数，只用 5~10% 的空间，却提升 3~5 倍的读取性能。

## Feed 流 Mc 架构

Feed 流的缓存体系中，对于 Memcached 存储采用 L1-Main-Backup 架构。这个架构前面在讲分布式 Memcached 实践中也有介绍。微博 Feed 流的 Memcached 存储架构体系中，L1 单池容量一般为 Main 池的 1/10，有 4~6 组 L1，用于存放最热的数据，可以很好的解决热点事件或节假日的流量洪峰问题。Main 池容量最大，保存了最近一段时间的几乎所有较热的数据。Backup 池的容量一般在 Main 池的 1/2 以下，主要解决 Main 池异常发生或者 miss 后的 key 访问。

L1-Main-Backup 三层 Memcached 架构，可以很好抵御突发洪峰流量、局部故障等。实践中，如果业务流量不大，还可以配置成两层 Main-Backup。对于 2 层或 3 层 Mc 架构，处理 Mc 指令需要各种穿透、回种，需要保持数据的一致性，这些策略相对比较复杂。因此微博构建了 proxy，封装 Mc 多层的读写逻辑，简化业务的访问。部分业务由于对响应时间很敏感，不希望因为增加 proxy 一跳而增加时间开销，因此微博也提供了对应的 client，由 client 获取并订阅 Mc 部署，对三层 Mc 架构进行直接访问。

在突发热点事件发生，大量用户上线并集中访问、发表 Feed，并且会对部分 Feed 进行超高并发的访问，总体流量增加 1 倍以上，热点数据所在的缓存节点流量增加数倍，此时需要能够快速增加多组 L1，从而快速分散这个节点数据的访问。另外在任何一层，如果有节点机器故障，也需要使用其他机器替代。这样三层 Mc 架构，时常需要进行一些变更。微博的 Mc 架构配置存放在配置中心 config-server 中，由 captain 进行管理。proxy、client 启动时读取并订阅这些配置，在 Mc 部署变更时，可以及时自动切换连接。

Feed 流处理程序访问 Mc 架构时，对于读请求，首先会随机选择一组 L1，如果 L1 命中则直接返回，否则读取 Main 层，如果 Main 命中，则首先将 value 回种到 L1，然后返回。如果 Main 层也 miss，就再读取 slave，如果 slave 命中，则回种 Main 和最初选择的那组 L1，然后返回。如果 slave 也 miss，就从 DB 加载后，回种到各层。这里有一个例外，就是 gets 请求，因为 gets 是为了接下来的 cas 更新服务，而三层 Mc 缓存是以 Main、Backup 为基准，所以 gets 请求直接访问 Main 层，如果 Main 层失败就访问 Backup，只要有一层访问获得数据则请求成功。后续 cas 时，将数据更新到对应 Main 或 Backup，如果 cas 成功，就把这个 key/value set 到其他各层。

对于数据更新，三层 Mc 缓存架构以 Main-Backup 为基准，即首先更新 Main 层，如果 Main 更新成功，则再写其他三层所有 Mc pool 池。如果 Main 层更新失败，再尝试更新 Backup 池，如果 Backup 池更新成功，再更新其他各层。如果 Main、Backup 都更新失败，则直接返回失败，不更新 L1 层。在数据回种，或者 Main 层更新成功后再更新其他各层时，Mc 指令的执行一般采用 noreply 方式，可以更高的完成多池写操作。

三层 Mc 架构，可以支撑百万级的 QPS 访问，各种场景下命中率高达 99% 以上，是 Feed 流处理程序稳定运行的重要支撑。

对于 Feed 流中的 Redis 存储访问，业务的 Redis 部署基本都采用 1 主多从的方式。同时多个子业务按类型分为 cluster 集群，通过多租户 proxy 进行访问。对于一些数据量很小的业务，还可以共享 Redis 存储，进行混合读写。对于一些响应时间敏感的业务，基于性能考虑，也支持 smart client 直接访问 Redis 集群。整个 Redis 集群，由 clusterManager 进行运维、slot 维护及迁移。配置中心记录集群相关的 proxy 部署及 Redis 配置及部署等。这个架构在之前的经典分布式缓存系统课程中有详细介绍，此处不再赘述。