17 | Cache：多级缓存架构在消息系统中的应用

2019-10-04 袁武林

即时消息技术剖析与实战 进入课程 >

讲述：袁武林
时长 15:30 大小 12.43M

你好，我是袁武林。

今天，我要带你了解的是一项在 IM 系统中相对比较通用的、使用比较高频的，而且对系统性能提升非常明显的技术：缓存。

说到缓存，你应该不陌生。相对于磁盘操作，基于内存的缓存对耗时敏感的高并发应用来说，在性能方面的提升是非常明显的。

下面是谷歌的技术奠基人杰夫·狄恩（Jeff Dean）给出的一些计算机相关的硬件指标，虽然有些数据可能由于时间太久不够准确，但大致的量级基本还是一致的。

L1 cache reference 0.5 ns

Branch mispredict 5 ns

L2 cache reference 7 ns

Mutex lock/unlock 100 ns

Main memory reference 100 ns

Compress 1K bytes with Zippy 10,000 ns

Send 2K bytes over 1 Gbps network 20,000 ns

Read 1 MB sequentially from memory 250,000 ns

Round trip within same datacenter 500,000 ns

Disk seek 10,000,000 ns

Read 1 MB sequentially from network 10,000,000 ns

Read 1 MB sequentially from disk 30,000,000 ns

Send packet CA->Netherlands->CA 150,000,000 ns

可以看到，同样是 1MB 的数据读取，从磁盘读取的耗时，比从内存读取的耗时相差近 100 倍，这也是为什么业界常说“处理高并发的三板斧是缓存、降级和限流” 了。

使用缓存虽然能够给我们带来诸多性能上的收益，但存在一个问题是缓存的资源成本非常高。因此，在 IM 系统中对于缓存的使用，就需要我们左右互搏地在“缓存命中率”和“缓存使用量”两大指标间不断均衡。

在今天的课程中，我会围绕 IM 系统中缓存的使用，来聊一聊在使用过程中容易碰到的一些问题及相应的解决方案。

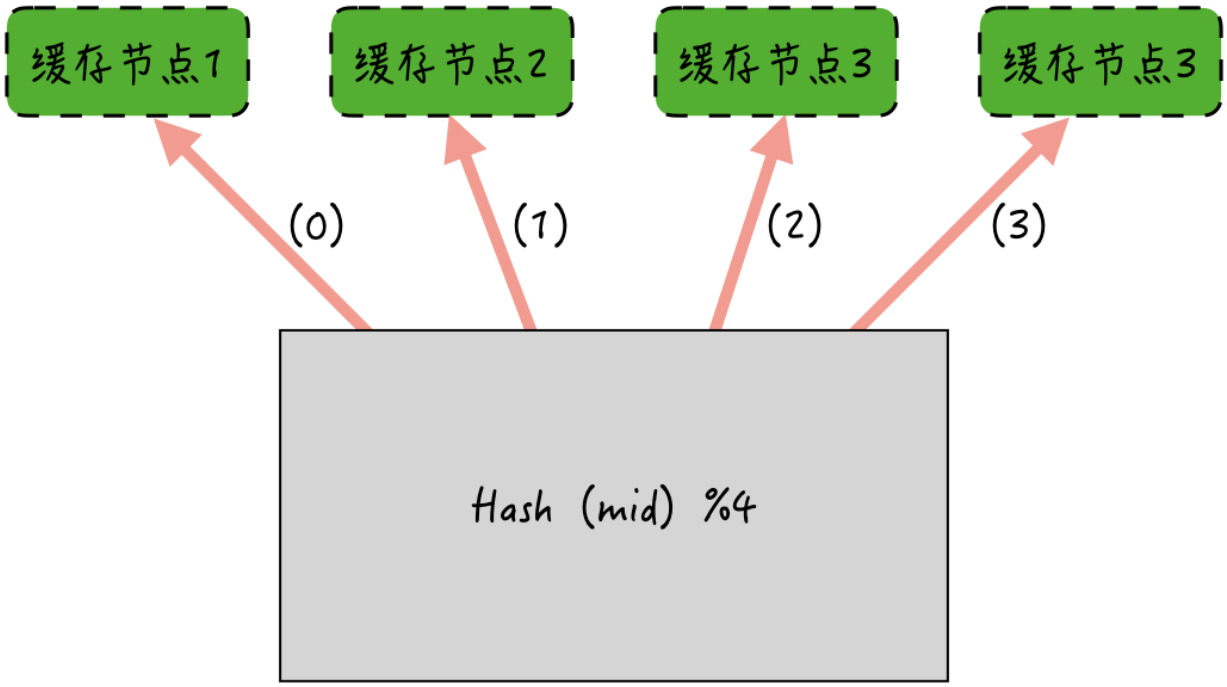
缓存的分布式算法

对于大规模分布式服务来说，大部分缓存的使用都是多实例分布式部署的。接下来，我们就先来了解一下缓存常见的两种分布式算法：取模求余与一致性哈希。

取模求余

取模求余的算法比较简单。比如说，用于存储消息内容的缓存，如果采用取模求余，就可以简单地使用消息 ID 对缓存实例的数量进行取模求余。

如下图所示：如果消息 ID 哈希后对缓存节点取模求余，余数是多少，就缓存到哪个节点上。



取模求余的分布式算法在实现上非常简单。但存在的问题是：如果某一个节点宕机或者加入新的节点，节点数量发生变化后，Hash 后取模求余的结果就可能和以前不一样了。由此导致的后果是：加减节点后，缓存命中率下降严重。

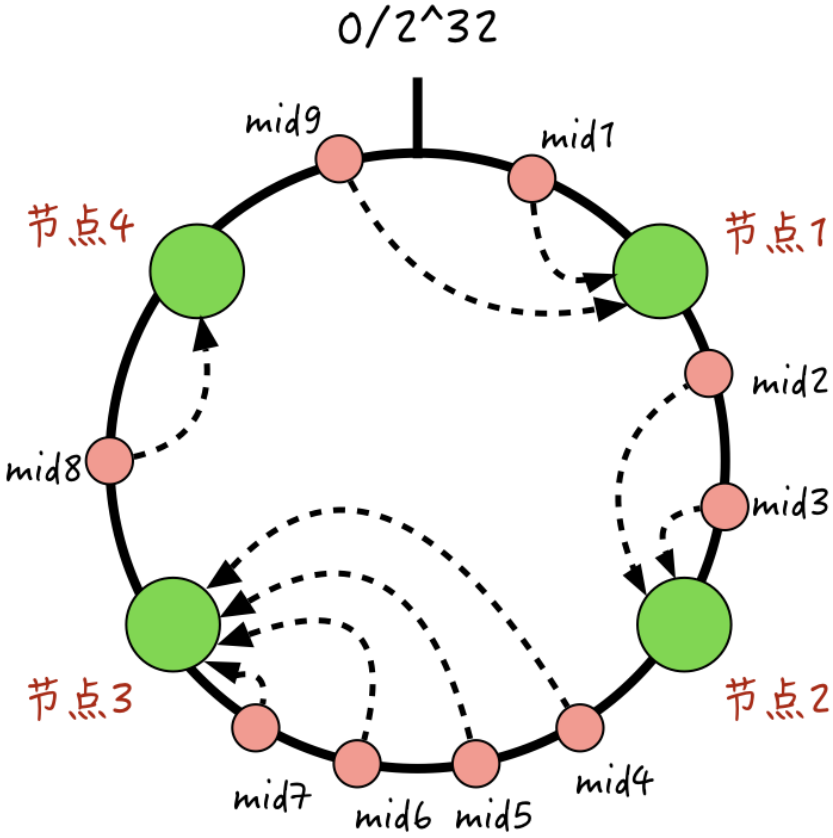
一致性哈希

为了解决这个问题，业界常用的另一种缓存分布式算法是一致性哈希。它是 1997 年麻省理工学院提出的一种算法，目前主要应用在分布式缓存场景中。

一致性哈希的算法是：把全量的缓存空间分成 2 的 32 次方个区域，这些区域组合成一个环形的存储结构；每一个缓存的消息 ID，都可以通过哈希算法，转化为一个 32 位的二进制数，也就是对应这 2 的 32 次方个缓存区域中的某一个；缓存的节点也遵循同样的哈希算法（比如利用节点的 IP 来哈希），这些缓存节点也都能被映射到 2 的 32 次方个区域中的某一个。

那么，如何让消息 ID 和具体的缓存节点对应起来呢？

很简单，每一个映射完的消息 ID，我们按顺时针旋转，找到离它最近的同样映射完的缓存节点，该节点就是消息 ID 对应的缓存节点。大概规则我画了一个图，你可以参考一下：



那么，为什么一致性哈希能够解决取模求余算法下，加减节点带来的命中率突降的问题呢?

结合上图，我们一起来看一下。假设已经存在了 4 个缓存节点，现在新增加一个节点 5，那么本来相应会落到节点 1 的 mid1 和 mid9，可能会由于节点 5 的加入，有的落入到节点 5，有的还是落入到节点 1；落入到新增的节点 5 的消息会被 miss 掉，但是仍然落到节点 1 的消息还是能命中之前的缓存的。

另外，其他的节点 2、3、4 对应的这些消息还是能保持不变的，所以整体缓存的命中率，相比取模取余算法波动会小很多。

同样，如果某一个节点宕机的话，一致性哈希也能保证，只会有小部分消息的缓存归属节点发生变化，大部分仍然能保持不变。

数据倾斜

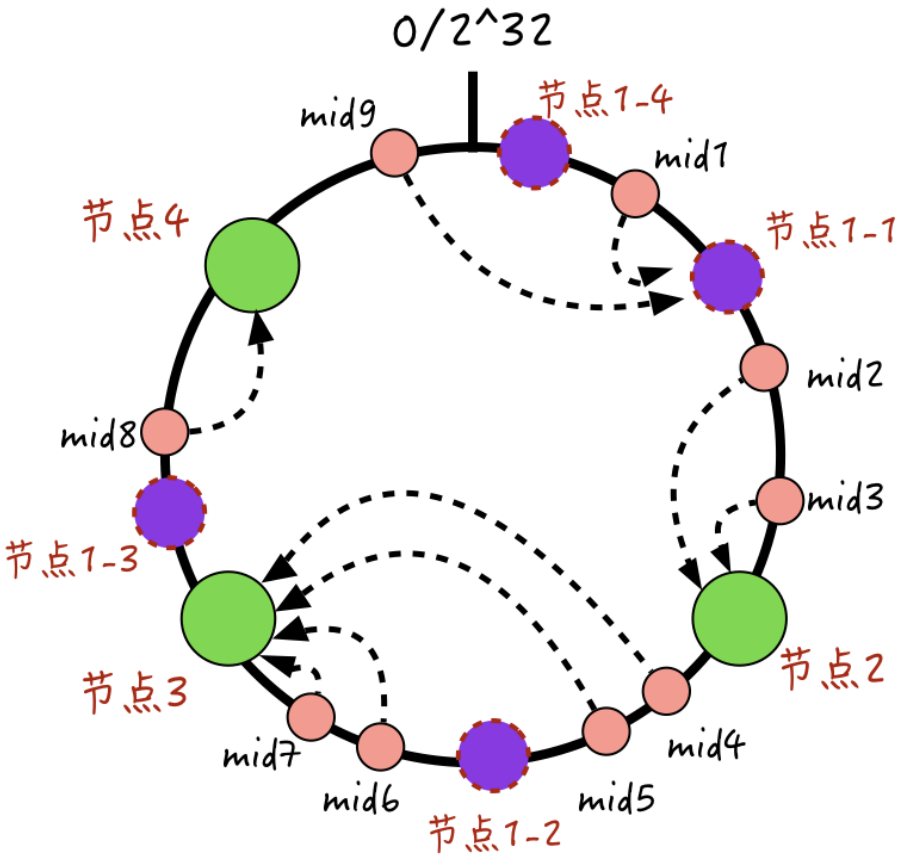
一致性哈希既然解决了加减节点带来的命中率下降的问题，那么是不是这种算法，就是缓存分布式算法的完美方案呢?

这里我们会发现，一致性哈希算法中，如果节点比较少，会容易出现节点间数据不均衡的情况，发生数据倾斜；如果节点很多，相应的消息就能在多个节点上分布得更均匀。

但在实际的线上业务中，部署的缓存机器节点是很有限的。

所以，为了解决物理节点少导致节点间数据倾斜的问题，我们还可以引入虚拟节点，来人为地创造更多缓存节点，以此让数据分布更加均匀。

虚拟节点的大概实现过程，你可以参考下图：



我们为每一个物理节点分配多个虚拟节点，比如在上图这里，给节点 1 虚拟出 4 个节点。当消息进行缓存哈希定位时，如果落到了这个物理节点上的任意一个虚拟节点，那么就表示，真正的缓存存储位置在这个物理节点上，然后服务端就可以从这个物理节点上进行数据的读写了。

如上面这个例子，本来都落在节点 3 的 4 条消息 mid4、mid5、mid6、mid7，在加入节点 1 的虚拟节点后，mid4 和 mid5 落到了虚拟节点 1-2 上，这样 mid4 和 mid5 就被分配到物理节点 1 上了。可见，通过这种方式，能更好地打散数据的分布，解决节点间数据不平衡的问题。

缓存热点问题

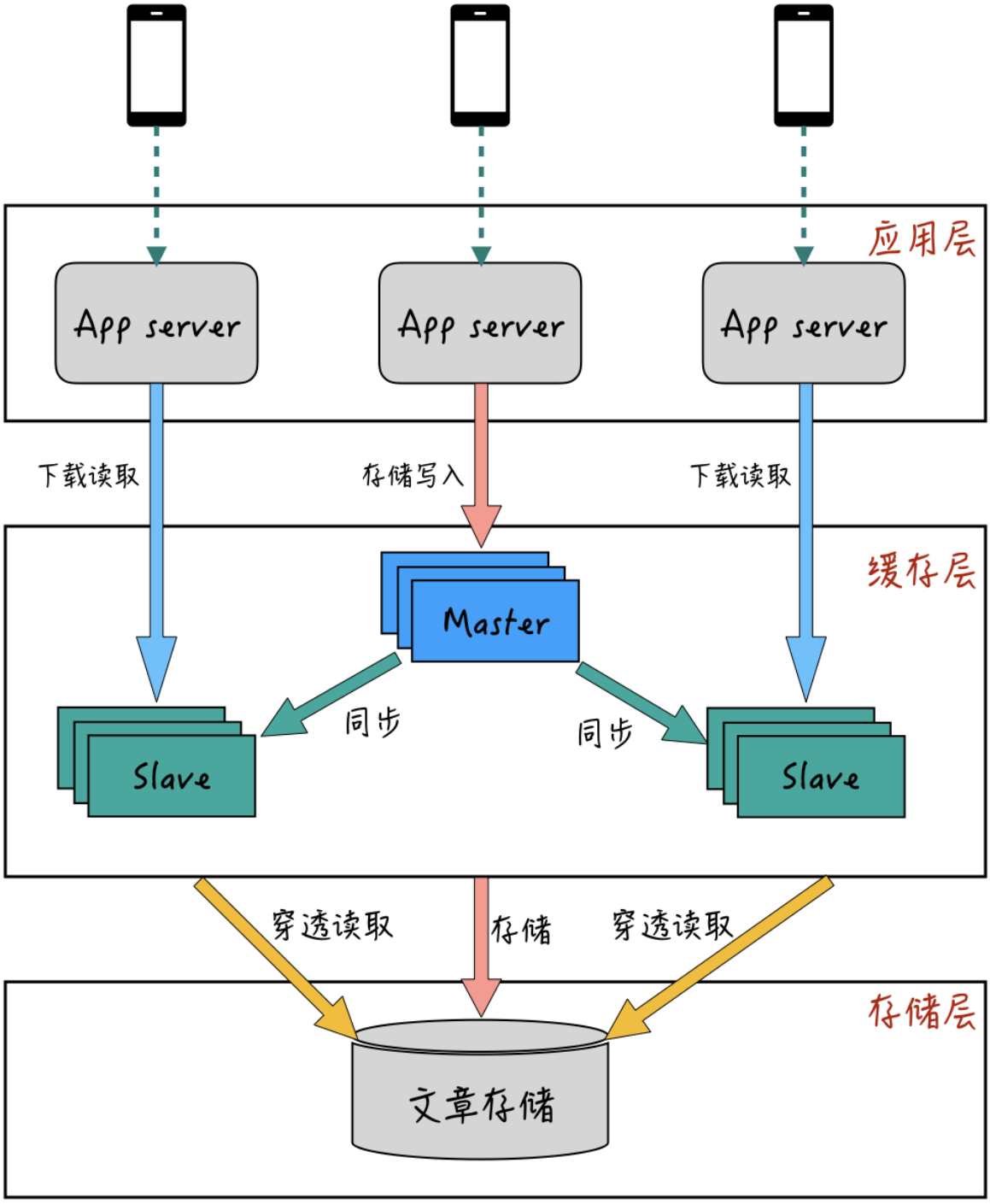
通过一致性哈希配合虚拟节点，我们解决了节点快速扩容和宕机，导致命中率下降的问题及节点间数据倾斜的问题。但在 IM 的一些场景里，还可能会出现单一资源热点的问题。

比如，一个超级大 V 给他的粉丝群发了一篇精心编写的长文章，可能一瞬间服务端会有上万的文章阅读请求涌入。由于这些长文章都是作为富文本进行存储的，所以存储的数据较大，有的文章都超过 1MB，而且用户还需要随时能够修改文章，也不好通过 CDN 来进行分发。

那么，我们如何去解决这种缓存热点问题呢？

多级缓存架构 - 主从模式

我以上面的“长文章流量热点”的例子来说明一下。为了防止文章下载阅读出现热点时，造成后端存储服务的压力太大，我们一般会通过缓存来进行下载时的加速。比如说，我们可以通过文章的唯一 ID 来进行哈希，并且通过缓存的一主多从模式来进行部署，主从模式的部署大概如下图：



一般来说，主从模式下，主库只用于数据写入和更新，从库只用于数据读取。当然，这个也不是一定的。

比如，在写多读少的场景下，也可以让主库承担一部分的数据读取工作。当缓存的数据读取 QPS 比较大的情况下，可以通过增加从库的方式来提升整体缓存层的抗读取能力。

主从模式是最常见的、使用最多的缓存应用模式。但是主从模式在某些突发流量的场景下会存在一些问题，就比如刚刚提到的“长文章流量热点”问题。

我们对某篇长文章的唯一 ID 来进行哈希，在主从模式下，一篇文章只会映射到一个从库节点上。虽然能够通过增加从库副本数来提升服务端对一篇文章的读取能力，但由于文章大小比较大，即使是多从库副本，对于千兆网卡的从库实例机器来说，带宽层面也很难抗住这个热点。举个例子，单台机器 120MB 带宽，对于 1MB 大小的文章来说，如果 QPS 到 1000 的话，至少需要 8 个实例才可以抗住。

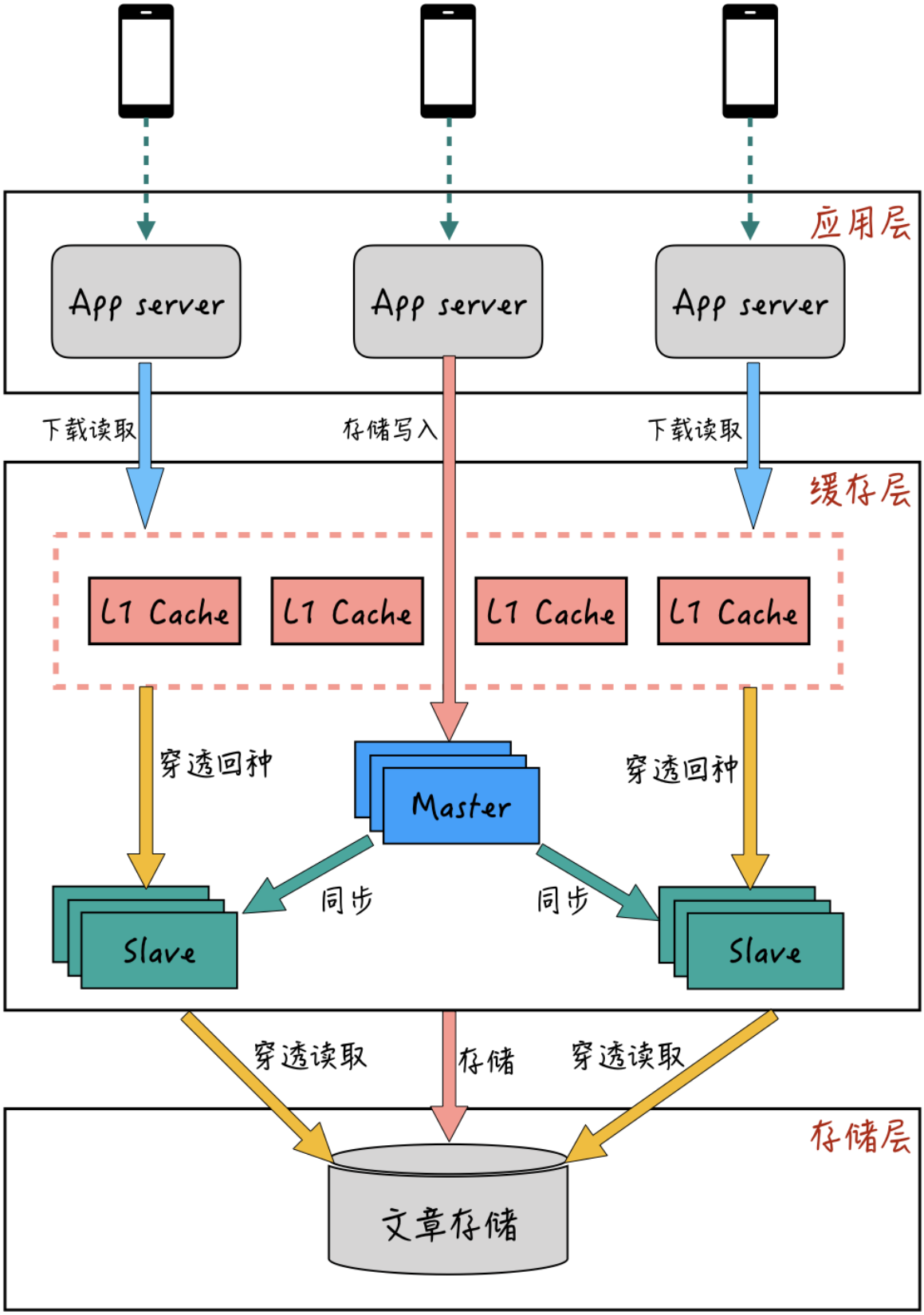
另外，多从库副本是对主库数据的完整拷贝，从成本上考虑也是非常不划算的。除了带宽问题，对于某些 QPS 很高的资源请求来说，如果采用的是单主单从结构，一旦从库宕机，瞬间会有大量请求直接穿透到 DB 存储层，可能直接会导致资源不可用。

多级缓存架构 -L1+ 主从模式

为了解决主从模式下，单点峰值过高导致单机带宽和热点数据在从库宕机后，造成后端资源瞬时压力的问题，我们可以参考 CPU 和主存的结构，在主从缓存结构前面再增加一层 L1 缓存层。

L1 缓存，顾名思义一般它的容量会比较小，用于缓存极热的数据。那么，为什么 L1 缓存可以解决主从模式下的带宽问题和穿透问题呢？

我们来看一下，L1+ 主从模式的部署和访问形式：



L1 缓存作为最前端的缓存层，在用户请求的时候，会先从 L1 缓存进行查询。如果 L1 缓存中没有，再从主从缓存里查询，查询到的结果也会回种一份到 L1 缓存中。

与主从缓存模式不一样的地方是：L1 缓存有分组的概念，一组 L1 可以有多个节点，每一组 L1 缓存都是一份全量的热数据，一个系统可以提供多组 L1 缓存，同一个数据的请求会轮流落到每一组 L1 里面。

比如同一个文章 ID，第一次请求会落到第一组 L1 缓存，第二次请求可能就落到第二组 L1 缓存。通过穿透后的回种，最后每一组 L1 缓存，都会缓存到同一篇文章。通过这种方式，同一篇文章就有多个 L1 缓存节点来抗读取的请求量了。

而且，L1 缓存一般采用 LRU（Least Recently Used）方式进行淘汰，这样既能减少 L1 缓存的内存使用量，也能保证热点数据不会被淘汰掉。并且，采用 L1+ 主从的双层模式，即使有某一层节点出现宕机的情况，也不会导致请求都穿透到后端存储上，导致资源出现问题。

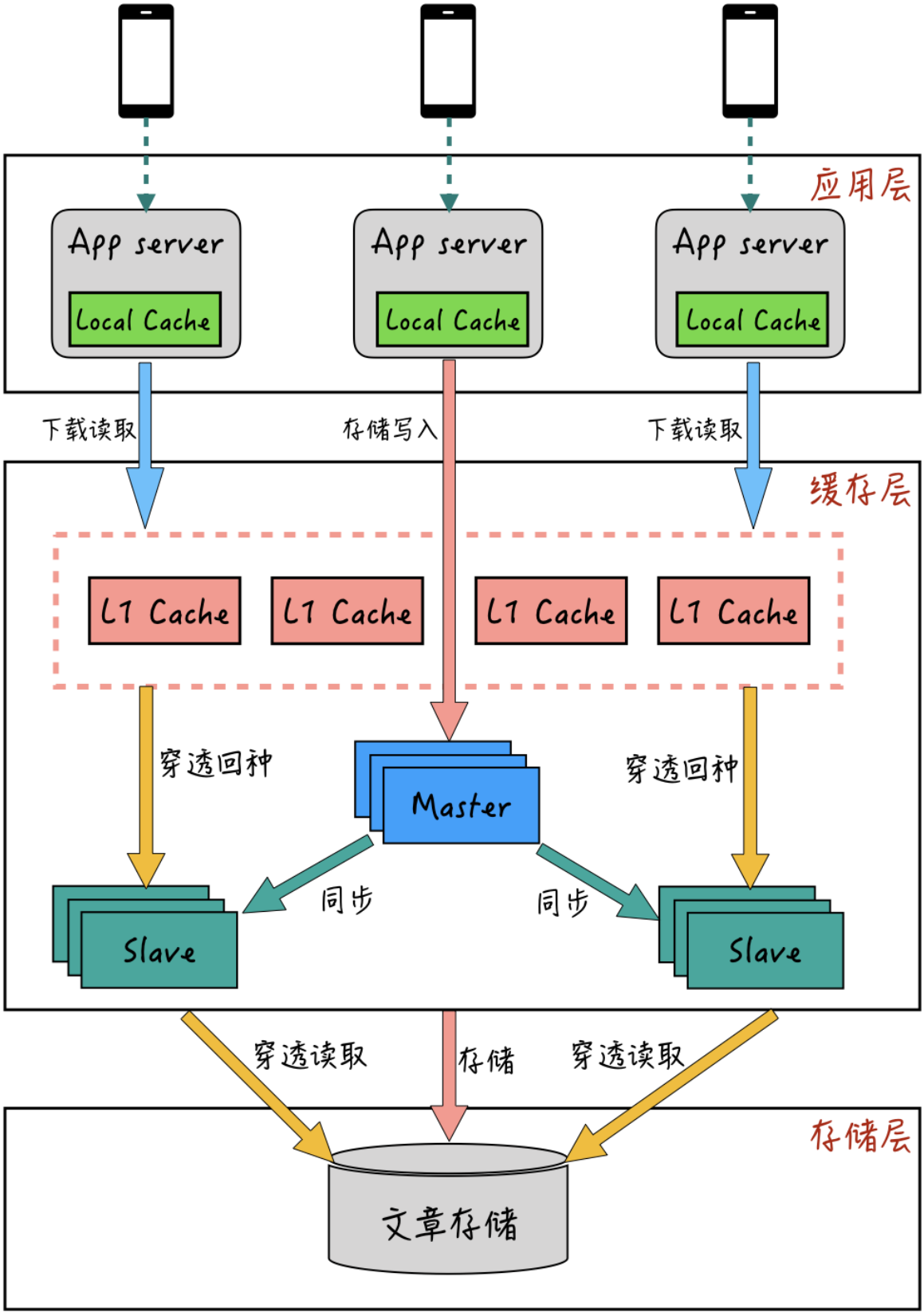
多级缓存架构 - 本地缓存 +L1+ 主从的多层模式

通过 L1 缓存 + 主从缓存的双层架构，我们用较少的资源解决了热点峰值的带宽问题和单点穿透问题。

但有的时候，面对一些极热的热点峰值，我们可能需要增加多组 L1 才能抗住带宽的需要。不过内存毕竟是比较昂贵的成本，所以有没有更好的平衡极热峰值和缓存成本的方法呢？

对于大部分请求量较大的应用来说，应用层机器的部署一般不会太少。如果我们的应用服务器本身也能够承担一部分数据缓存的工作，就能充分利用应用层机器的带宽和极少的内存，来低成本地解决带宽问题了。那么，这种方式是否可以实现呢？

答案是可以的，这种本地缓存 +L1 缓存 + 主从缓存的多级缓存模式，也是业界比较成熟的方案了。多级缓存模式的整体流程大概如下图：



本地缓存一般位于应用服务器的部署机器上，使用应用服务器本身的少量内存。它是应用层获取数据的第一道缓存，应用层获取数据时先访问本地缓存，如果未命中，再通过远程从 L1 缓存层获取，最终获取到的数据再回种到本地缓存中。

通过增加本地缓存，依托应用服务器的多部署节点，基本就能完全解决热点数据带宽的问题。而且，相比较从远程 L1 缓存获取数据，本地缓存离应用和用户设备更近，性能上也会更好一些。

但是使用本地缓存有一个需要考虑的问题，那就是数据的一致性问题。

还是以“长文章”为例。我们的服务端可能会随时接收到用户需要修改文章内容的请求，这个时候，对于本地缓存来说，由于应用服务器的部署机器随着扩缩容的改变，其数量不一定是固定的，所以修改后的数据如何同步到本地缓存中，就是一个比较复杂和麻烦的事情了。

要解决本地缓存一致性问题，业界比较折中的方式是：对本地缓存采用“短过期时间”的方式，来平衡本地缓存命中率和数据更新一致性的问题。比如说，针对“长文章”的本地缓存，我们可以采用 5 秒过期的策略，淘汰后再从中央缓存获取新的数据。这种方式对于大部分业务场景来说，在产品层面上也是都能接受的。

小结

好了，下面简单回顾一下今天课程的内容。

首先，我介绍了缓存在高并发应用中的重要性，以及在 IM 系统中使用的部分场景。然后再带你了解了缓存分布式的两种算法：取模求余和一致性哈希。

> **取模求余算法**在实现上非常简单，但存在的问题是，取模求余算法在节点扩容和宕机后会出现震荡，缓存命中率会严重降低。
>
> **一致性哈希算法**解决了节点增删时震荡的问题，并通过虚拟节点的引入，缓解了**“数据倾斜”**的情况。

最后，我着重介绍了业界通用的三种分布式缓存的常见架构。

> **一种是主从模式。**简单的主从模式最常见，但是在面对峰值热点流量时，容易出现带宽问题，也存在缓存节点宕机后穿透到存储层的问题。
>
> **第二种是 L1+ 主从模式。**通过增加 L1 缓存层，以并行的多组小容量的 L1 缓存，解决了单一热点的带宽问题，也避免了单一节点宕机后容易穿透到 DB 存储层的情况。
>
> **最后一种是本地缓存 +L1+ 主从的多层模式。**作为低成本的解决方案，我们在 L1+ 主从模式的基础上，引入了本地缓存。本地缓存依托应用服务器的本机少量内存，既提升了资

源的有效利用，也彻底解决了带宽的问题。同时在性能方面，也比远程缓存获取更加优秀。对于本地缓存的数据一致性问题，我们可以通过“短过期时间”来平衡缓存命中率和数据一致性。

面对高并发业务带来的流量压力，我们不可否认的是，缓存的使用是目前为止最有效的提升系统整体性能的手段。作为系统优化的一把利器，如何用好这个强大的工具，是你需要去不断思考和学习的。希望今天介绍的这几种缓存使用的姿势，能够让你有所收获，并能在自己的业务中去尝试实践。

最后给你留一道思考题：

L1+ 主从模式下，如果热点数据都被 L1 缓存层拦截命中，会导致主从缓存层相应的这个热点数据，由于长时间得不到读取而被 LRU 淘汰掉。这样，如果下线 L1 缓存，还是会有不少的请求直接穿透到 DB 存储层。那么有没有办法，能够让主从缓存在有 L1 缓存层的情况下，依旧能保持数据热度？

以上就是今天课程的内容，欢迎你给我留言，我们可以在留言区一起讨论，感谢你的收听，我们下期再见。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　16 | APNs：聊一聊第三方系统级消息通道的事

下一篇　18 | Docker容器化：说一说IM系统中模块水平扩展的实现

精选留言 (10)

写留言

小小小丶盘子
2019-10-04

开线程定时访问热点数据，保证不被淘汰。

展开

作者回复: 也是一种思路，不过实现上可能会比较复杂。另外可以考虑把master也加入到L1缓存层中，这样能保持数据热度。

1

A:春哥大魔王
2019-10-07

老师L1缓存具体是实现和缓存层究竟有什么区别呢？还是说技术实现上本身没有区别，只不过存储目的不同？比如L1就是为了解决热数据的

展开

那时刻
2019-10-06

请问怎么保证local cache到cache master的数据一致性呢？毕竟local cache在分布式的服务器集群，会有同一个文档发到不同服务器local cache的情形。从图中看出，是某一个local cache负责同步数据到cache master么？或者我有什么误解？烦请指正

展开

Geek_e986e3
2019-10-06

老师 我看你说将master节点当作l1来用保证不过期。这样的话我能理解成l1实际上也就是容量缩小版的从节点吗？

展开

卫江
2019-10-06

思考题，我感觉可以在L1层，根据访问的特定数据的频率，比如qps超过100就去后面缓存拉一下，这样一来实现简单，频率不高又能保证热数据不被淘汰，同时也可以作为L1缓存的数据更新保证数据一致性的实现方式。

展开


_CountingStars
2019-10-05

主从缓存层不使用lru算法淘汰数据 保存全量数据 内存不够 就使用SSD存储数据 比如 ssdb rocksdb leveldb


探索无止境
2019-10-05

老师理论都已经清楚了，能否提供一个可落地的具体实现，比如L1缓存具体要怎么做？


孙凯
2019-10-04

老师能详细讲下L1级缓存怎么用么？

展开

作者回复: 实现上并不复杂，其实就是二次哈希的过程，比如将原来哈希到slave1的请求再采用round robin轮流打到多组L1上，这样就实现流量分散了。


刘丹
2019-10-04

请问L1缓存是指redis、memcached这种网络缓存吗？

展开

作者回复: 一般真实场景中来说是的，不过L1主要还是一种缓存架构，实现上没有限制的。


东东

2019-10-04

老师，hash(uid)%10和uid%10这两个有啥区别？然后L1空间很小，一般存放哪些类型的数据呢？

作者回复: 有的uid是字符型的，所以通用做法就是先hash一下。
L1缓存一般自动lru成最热的数据。