

39 | 存储与缓存

2019-09-06 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 12:18 大小 11.28M



你好，我是七牛云许式伟。

前面接连三讲我们介绍了存储中间件的由来，以及最为常见的存储中间件，如：键值存储（KV Storage）、数据库（Database）、对象存储（Object Storage）。

当然，它们并不是全部。常见的存储中间件还有很多，比如消息队列（MQ）、搜索引擎（Search Engine）等等。

限于篇幅，我们不能——对它们进行分析。今天，我们聊一聊缓存（Cache）。

memcached


缓存 (Cache) 是什么?

简单说, 缓存是存储 (Storage) 的加速器。加速的原理通常是这样几种方法:

最常见的是用更高速的硬件来加速。比如, 用 SSD 缓存加速 SATA 存储, 用内存缓存加速基于外存的存储。

还有一种常见的方法是用更短的路径。比如, 假设某个计算 $y = F(x)$ 非常复杂, 中间涉及很多步骤, 发生了一系列的存储访问请求, 但是这个计算经常会被用到, 那么我们就可以用一个 $x \Rightarrow y$ 的内存缓存来加速。


可见, 缓存的数据结构从实现上来讲只需要是一个键值存储。所以它的接口可以非常简单:

 复制代码

```
1 type Cache {
2     ...
3 }
4
5 func (cache *Cache) Get(key []byte) (val []byte, err error)
6 func (cache *Cache) Set(key, val []byte) (err error)
7 func (cache *Cache) Delete(key []byte) (err error)
```

第一个被广泛应用的内存缓存是 memcached。通常, 我们会使用多个 memcached 实例构成一个集群, 通过 Hash 分片或者 Range 分片将缓存数据分布到这些实例上。

一个典型的 memcached 的使用方式如下:

 复制代码

```
1 func FastF(x TypeX) (y TypeY) {
2     key := toBytes(x)
3     hash := hashOf(key)
4     i := hash % countOf(memcaches)
5     val, err := memcaches[i].Get(key)
6     if err != nil {
7         y = F(x)
8         val = toBytes(y)
9         memcaches[i].Set(key, val)
10    } else {
11        y = fromBytes(val)
```

```
12     }  
13     return  
14 }
```

类似的缓存逻辑大家应该比较经常见到。

这个示例我们采用的是简单 Hash 分片的方法，它的好处是非常容易理解。当然不太好的地方在于，一旦我们要对 memcached 集群扩容，countOf(memcachedes) 就会变化，导致大量的 key 原先落在某个分片，现在就落到一个新的分片。

这会导致大量的缓存未命中（Cache Miss），也就是 cache.Get(key) 返回失败。在缓存未命中的情况下，FastF(x) 不只是没有加速 F(x)，还增加了两次网络请求：cache.Get 和 cache.Set。

所以缓存系统的一个核心指标是缓存命中率（Cache Hit Rate），即在一段时间内，FastF 缓存命中的次数 / 所有 FastF 的调用次数。

为了避免 memcached 集群扩容导致缓存命中率大幅降低，一般我们不会用简单哈希分片，而是用一致性哈希。

什么情况下需要扩容？一旦缓存命中率趋势下降，且下降到某个阈值，就要考虑给缓存集群扩容。

缓存 vs 存储

通过以上的介绍可以看出，缓存的基础逻辑是非常简单的。问题是：

缓存（Cache）和存储（Storage）是什么关系？它也是一种存储中间件么？

既是也不是。

首先，缓存和一般的存储中间件一样，也在维持着业务状态。从这个角度看，缓存的确是一类存储。

但是，缓存允许数据发生丢失，所以缓存通常是单副本的。一个内存缓存的集群挂了一个实例，或者一个外存缓存的集群坏了一块硬盘，单就缓存集群本身而言，就出现数据丢失。

缓存数据丢失，这事可大可小。只要不是发生大片大片的缓存数据丢失的情形，通常只是会造成后端存储（Storage）的短时压力变大。

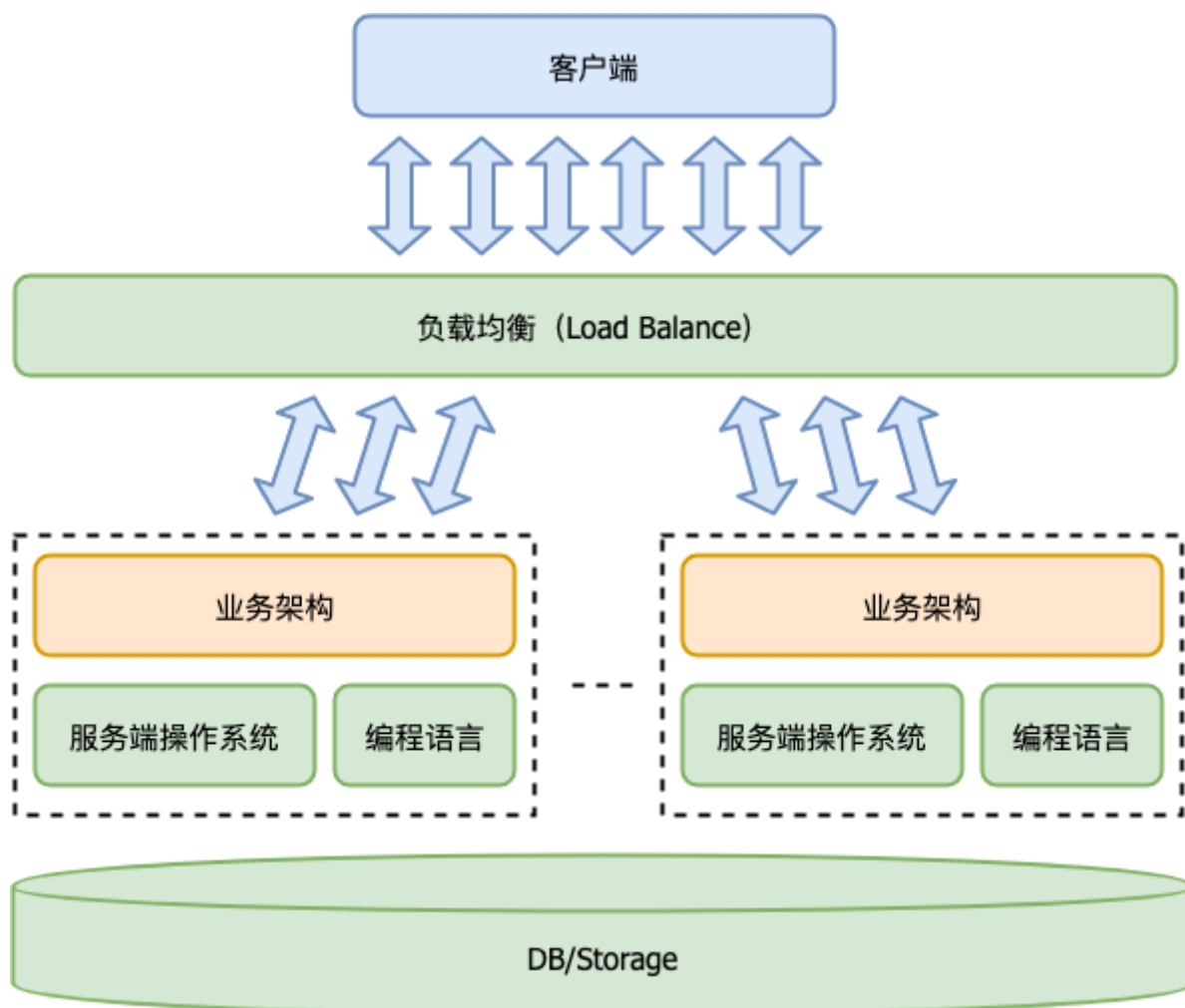
但在极端的情况下，可能会出现雪崩的情况。

雪崩怎么形成？首先是部分缓存实例宕机，导致缓存命中率（Cache Hit Rate）下降，大量的请求落到后端存储上，导致后端存储过载，也出现宕机。

这时就会出现连锁反应，形成雪崩现象。后端存储就算重新启动起来，又会继续被巨大的用户请求压垮，整个系统怎么启动也启动不了。

应该怎么应对雪崩？最简单的办法，是后端存储自己要有过载保护能力。一旦并发的请求超过预期，就要丢弃部分请求，以减少压力。

我们在本章开篇第一讲 [“34 | 服务端开发的宏观视角”](#) 中，总结服务端开发的体系架构如下：



在这个图中，我们并没有把缓存（Cache）画出来。但结合上面介绍的缓存典型使用方式，我们很容易脑补它在图中处于什么样的位置。

回到前面的问题，缓存（Cache）和存储（Storage）到底是什么关系？

我个人认为，缓存其实应该被认为是存储的补丁，而且是理论上来说不太完美的补丁。

为什么说它是补丁？

因为如果存储本身非常匹配业务场景的话，它不应该需要缓存在它前面挡一道，内部自己就有缓存。至于把一个复杂的 $F(x)$ 缓存起来，更根本的原因还是存储和业务场景不那么直接匹配所致。

但是实现一个存储很难，所以存储的业务场景匹配性很难做到处处都很好。

出现事务（Transaction），是为了改善存储的业务场景“写操作”的匹配性，把一个复杂操作包装成一个原子操作。

出现缓存 (Cache)，则是为了改善存储的业务场景“读操作”的匹配性，提升高频读操作的效率。

所以我们说，缓存是一个存储的补丁。

那么为什么我们说这是一个不太完美的补丁呢？

因为上面的 $\text{FastF}(x)$ 并没有被包装成一个原子的读操作。从严谨的角度来说，这段代码逻辑是有问题的，它会破坏数据的一致性。

对于一个确定的 x 值，如果 $F(x)$ 永远不变，这就没问题。但如果 $F(x)$ 值会发生变化，会有多个版本的值，那就有可能出现并发的两个 $F(x)$ 请求得到的结果不同，从而导致缓存中的值和存储中的值不一致。

这种情况后果有可能会比较严重。尤其是如果我们有一些业务逻辑是基于 $\text{FastF}(x)$ 得到的值，就有可能出现逻辑错乱。

groupcache

为了避免发生这类一致性问题，memcached 的作者 Brad Fitzpatrick (bradfitz) 搞了一个新的内存缓存系统，叫 groupcache。

groupcache 基于 Go 语言实现，其 Github 主页为：

<https://github.com/golang/groupcache>

从业务角度，groupcache 主要做了两大变化：

其一，引入 group 的概念。这是一个重要改动，也是 groupcache 这个名字的来由。

在同一个缓存集群，可能会需要缓存多个复杂操作，比如 $F(x)$ 、 $G(x)$ 。如果没有 group，那么我们就不能只是记录 $x \Rightarrow y$ 这样的键值对，而是要记录 $F\#x \Rightarrow y$ ， $G\#x \Rightarrow y$ 这样的键值对。中间的 # 只是一个分隔符，换其他的也可以。

看起来好像也还可以？


其实不然，因为 $F(x)$ 、 $G(x)$ 在同一个内存缓存集群就意味着它们相互之间会淘汰对方，这里的淘汰规则不是我们能够控制的，很难保证结果符合我们的预期。

那么有 group 会变成什么样？首先你可以创建 F 、 G 两个独立的 group，每个 group 可以设定独立的内存占用上限（cacheBytes）。

这样，每个 group 就只淘汰自己这个 group 内的数据，相当于有多个逻辑上独立的内存缓存集群。

另外，在 group 中只需要记录 $x \Rightarrow y$ 这样的键值对，不再需要用 $F\#x$ 、 $G\#x$ 这种手工连接字符串的方式来模拟出名字空间。

其二，值不可修改。一旦某个 x 值 Get 到的值为 y ，那么就一直为 y 。它的使用方式大体如下：

 复制代码

```
1 var groupF = groupcache.NewGroup("F", cacheBytes, groupcache.GetterFunc(func(ctx groupc
2   x := fromString(key)
3   y := F(x)
4   return dest.SetBytes(toBytes(y))
5 })))
6
7 func FastF(x TypeX) (y TypeY) {
8   key := toString(x)
9   var val []byte
10  groupF.Get(ctx, key, groupcache.AllocatingByteSliceSink(&val))
11  y = fromBytes(val)
12  return
13 }
```

这当然也就意味着它也不需要引入 memcached 中的缓存失效时间这样的概念。因为值是不会过时的，它只会因为内存不足而被淘汰。

一致性问题也被解决了。既然值不可修改，那么自然就不存在一致性问题。

当然，groupcache 是一个理论完美的内存缓存系统，它解决了 memcached 存在的一致性缺陷。但是 groupcache 对使用者来说是有挑战的，某种意义上来说，它鼓励我们用函数式编程的方式来实现业务逻辑。

但是你也知道，函数式编程是比较小众的。所以怎么用好 groupcache，挑战并不低。

Redis

谈到存储与缓存的关系，不能不提 Redis。

Redis 在定位上特别奇怪，以至于不同的人对它的认知并不相同。有的人会认为它是内存缓存，有的人会认为它是存储。

Redis 的确可以当作缓存来用，我们可以设置内存上限，当内存使用达到上限后，Redis 就会执行缓存淘汰算法。只不过，如果我们把它当作内存缓存，那么其实它只需要是一个简单的键值存储（KV Storage）就行。

但是 Redis 实际上是 key => document，它的值可以是各类数据结构，比如：字符串，哈希表，列表，集合，有序集合（支持 Range 查询），等等。

不仅如此，Redis 还支持执行 Lua 脚本来做存储过程。

这些都让 Redis 看起来更像一个数据库类的存储中间件。

但当我们把 Redis 看作存储，我们有这样一些重要的问题需要考虑。这些问题非常非常重要，存储系统可不是闹着玩的。

问题一，是持久性（Durability）。Redis 毕竟是基于内存的存储，虽然它也支持定期写到外存中，但是定期持久化的策略对于一个服务端的存储系统来说是不合格的。因为如果发生宕机，上一次持久化之后的新数据就丢了。

所以 Redis 需要其他的提升持久性的方案，比如多副本。

Redis 的确支持多副本。但是只是同机房多台机器的多副本是没有用的，因为它没有办法防止机房整体断电这类的故障。当出现机房级的故障时，就有极大概率会丢失数据。

对于存储系统来说，这是不可接受的。因为相比人们对持久性的要求，机房整体断电并不是一个太小概率的事件。

所以 Redis 如果要作为存储的话，必须保证用多机房多副本的方式，才能保证在持久性这一点上能够达标。

但是多机房多副本这样的方式，显然实施条件过于苛刻。会有多少企业仅仅是为了部署 Redis 去搞多个机房呢？

问题二，是重试的友好性。在 [“29 | 实战（四）：怎么设计一个“画图”程序？”](#) 中提到过，考虑网络的不稳定性，我们设计网络协议的时候需要考虑重试的友好性。

在 Redis 的协议中，有不少请求用户很友好，但是对重试并不友好。比如，LPUSH 请求用来给列表（List）增加一个元素。但是在重试时一个不小心，我们很可能就往列表中添加了多个相同的元素进去。

总结来说，Redis 如果我们把它作为存储的话，坑还是不少的。它和 memcached 都是实用型的瑞士军刀，很有用，但是我们站在分布式系统的理论角度看时，它们都有那么一点不完美的地方。

结语

今天我们讨论了存储与缓存之间的关系，也分别介绍了三个模型迥异的缓存系统：memcached、groupcache、Redis。

缓存是一个存储系统在服务器性能上的补丁。这个补丁并不是那么完美。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。我们服务端开发相关的基础软件介绍得差不多了，下一讲我们将聊聊服务端开发的架构建议。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

许式伟的架构课

从源头出发, 带你重新理解架构设计

许式伟
七牛云 CEO



新版升级: 点击「 请朋友读」, 20位好友免费读, 邀请订阅更有**现金**奖励。

© 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 38 | 文件系统与对象存储

精选留言 (10)

写留言



Geek_88604f

2019-09-07

对于一个确定的 x 值, 如果 $F(x)$ 永远不变, 这就没问题。但如果 $F(x)$ 值是不确定的, 那就有可能出现并发的两个 $F(x)$ 请求得到的结果不同, 从而导致缓存中的值和存储中的值不一致。

这段描述我是这么理解的, 老师的意思是不是两个线程同时去get同一个key, 发现key不在缓存中, 此时两个线程都会去计算key对应的value, 当线程A拿到x完成计算后准备...
展开

作者回复: 你的理解是对的, 当存在并行的Set请求, 自然存在时序问题, 导致存储和缓存数据不一致。解决方案来说:

- 1、sleep不能解决多任务协同问题, 所以这个方案不可行。
- 2、是可行的思路, 把 $F(x)$ 和 Set 一起串行执行。不过这会导致在缓存未命中时 $F(x)$ 执行两遍。你说的缓存未命中从存储读, 本质上是执行 $F(x)$ 的意思。
- 3、存储中 x 对应的数据发生变化时, 我们通常的做法是把 x 从缓存中清除 (Delete), 而不是执

行 F(x) 和 Set。原因是缓存空间是有限的，所以要给 Get 次数比较多的数据缓存，而不是一发生变更就缓存，这样非常可能反而降低了缓存命中率。

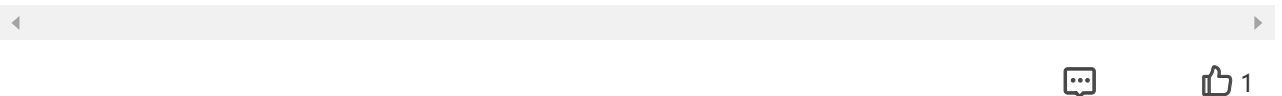


Charles

2019-09-08

所以许老师怎么去评估一个系统是否应该上缓存，假设目前存储都可以顶住负载?谢谢

作者回复: 要分析清楚压力和效率瓶颈。压力大了，可以考虑加缓存，当然也可以考虑存储扩容。效率瓶颈，通常只能用缓存解决。



Aaron Cheung

2019-09-09

groupcache 学习了 打卡39

展开 ▾



醉雪飘痕

2019-09-07

不好意思，突然明白了，应该是指缓存满进行的淘汰。而分不同的group后，各个group不相互影响，可独立进行淘汰，控制粒度更细。
请忽略我上面的提问，谢谢。



醉雪飘痕

2019-09-07

“F(x)、G(x) 在同一个内存缓存集群就意味着它们相互之间会淘汰对方？”
许老师，这里有些不理解，为何F(x)、G(x)会相互淘汰对方？



Geek_88604f

2019-09-06

```
func FastF(x TypeX) (y TypeY) {  
    key := toBytes(x)  
    hash := hashOf(key)  
    i := hash % countOf(memcaches)
```

```
val, err := memcaches[i].Get(key)...
```

展开 ▾

作者回复: err != nil 表示失败

leslie

2019-09-06

课程讲到现在算是明白为何redis、memcache的考虑因素了：之前很多地方的讲解没有涉及到底层算法，故而让我们觉得好像类似，但是实际上完全不同的；原来是忽略了底层算法。

老师今天的课从不一样的角度去解释：彻底明白为何差不多的东西其实各种称呼方式，大多数情况下其实很多时候没有强调 缓存与内存，这个概念被统称了。谢谢老师的...

展开 ▾

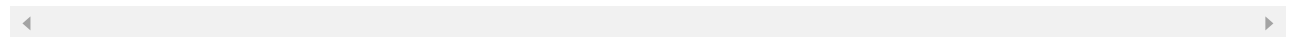


诗泽

2019-09-06

如果 $F(x)$ 值是不确定的，这种情况下放缓存里也就没意义了吧？

作者回复: 这里说的不确定不是太准确，正确应该说会改变，有多个版本



CoderLim

2019-09-06

缓存热数据可以有效缓解存储的压力，提高响应速度，但是设计时需要考虑扩缩容是否影响 hash 映射，是否重试友好，是否有持久性的需求

展开 ▾



风清扬

2019-09-06

许老师，缓存血崩的原因是命中率降低，大量请求直达后端，后端性能极速下降导致，解决办法是抛弃过多的请求。想到的一个是应用层既网关层限流。能详细讲解下吗？

作者回复: 后面还会细聊这些问题

