02 | 缓存一致：读多写少时，如何解决数据更新缓存不同步？

2022-10-26 徐长龙 来自北京

《高并发系统实战课》

讲述：徐长龙

时长 14:32　大小 13.28M

你好，我是徐长龙，我们继续来看用户中心性能改造的缓存技巧。

上节课我们对数据做了归类整理，让系统的数据更容易做缓存。为了降低数据库的压力，接下来我们需要逐步给系统增加缓存。所以这节课，我会结合用户中心的一些业务场景，带你看看如何使用临时缓存或长期缓存应对高并发查询，帮你掌握高并发流量下缓存数据一致性的相关技巧。

我们之前提到过，互联网大多数业务场景的数据都属于读多写少，在请求的读写比例中，写的比例会达到百分之一，甚至千分之一。

而对于用户中心的业务来说，这个比例会更大一些，毕竟用户不会频繁地更新自己的信息和密码，所以这种读多写少的场景特别适合做读取缓存。通过缓存可以大大降低系统数据层的查询压力，拥有更好的并发查询性能。但是，使用缓存后往往会碰到更新不同步的问题，下面我们具体看一看。

缓存性价比

缓存可以滥用吗？在对用户中心优化时，一开始就碰到了这个有趣的问题。 

就像刚才所说，我们认为用户信息放进缓存可以快速提高性能，所以在优化之初，我们第一个想到的就是将用户中心账号信息放到缓存。这个表有 2000 万条数据，主要用途是在用户登录时，通过用户提交的账号和密码对数据库进行检索，确认用户账号和密码是否正确，同时查看账户是否被封禁，以此来判定用户是否可以登录：

复制代码

```
# 表结构
CREATE TABLE `accounts` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `account` varchar(15) NOT NULL DEFAULT '',
  `password` char(32) NOT NULL,
  `salt` char(16) NOT NULL,
  `status` tinyint(3) NOT NULL DEFAULT '0'
  `update_time` int(10) NOT NULL DEFAULT '0',
  `create_time` int(10) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicod

# 登录查询
select id, account, update_time from accounts
where account = 'user1'
and password = '6b9260b1e02041a665d4e4a5117cfe16'
and status = 1
```

这是一个很简单的查询，你可能会想：如果我们将 2000 万的用户数据放到缓存，肯定能提供性能很好的服务。

这个想法是对的，但不全对，因为它的性价比并不高：这个表查询的场景主要用于账号登录，用户即使频繁登录，也不会造成太大的流量冲击。因此，缓存在大部分时间是闲置状态，我们没必要将并发不高的数据放到缓存当中，浪费我们的预算。

这就牵扯到了一个很核心的问题，**我们做缓存是要考虑性价比的**。如果我们费时费力地把一些数据放到缓存当中，但并不能提高系统的性能，反倒让我们浪费了大量的时间和金钱，那就是不合适的。我们需要评估缓存是否有效，一般来说，只有热点数据放到缓存才更有价值。

临时热缓存

推翻将所有账号信息放到缓存这个想法后，我们把目标放到会被高频查询的信息上，也就是用户信息。

用户信息的使用频率很高，在很多场景下会被频繁查询展示，比如我们在论坛上看到的发帖人头像、昵称、性别等，这些都是需要频繁展示的数据，不过这些数据的总量很大，全部放入缓存很浪费空间。

对于这种数据，我建议使用临时缓存方式，就是在用户信息第一次被使用的时候，同时将数据放到缓存当中，短期内如果再次有类似的查询就可以快速从缓存中获取。这个方式能有效降低数据库的查询压力。常见方式实现的**临时缓存**的代码如下：

复制代码

```go
// 尝试从缓存中直接获取用户信息
userinfo, err := Redis.Get("user_info_9527")
if err != nil {
  return nil, err
}

//缓存命中找到，直接返回用户信息
if userinfo != nil {
  return userinfo, nil
}

//没有命中缓存，从数据库中获取
userinfo, err := userInfoModel.GetUserInfoById(9527)
if err != nil {
  return nil, err
}

//查找到用户信息
if userinfo != nil {
  //将用户信息缓存，并设置TTL超时时间让其60秒后失效
  Redis.Set("user_info_9527", userinfo, 60)
  return userinfo, nil
}

// 没有找到，放一个空数据进去，短期内不再问数据库
// 可选，这个是用来预防缓存穿透查询攻击的
Redis.Set("user_info_9527", "", 30)
return nil, nil
```

可以看到，我们的数据只是临时放到缓存，等待 60 秒过期后数据就会被淘汰，如果有同样的数据查询需要，我们的代码会将数据重新填入缓存继续使用。这种临时缓存适合表中数据量

大，但热数据少的情况，可以降低热点数据的压力。

而之所以给缓存设置数据 TTL，是为了节省我们的内存空间。当数据在一段时间内不被使用后就会被淘汰，这样我们就不用购买太大的内存了。这种方式相对来说有极高的性价比，并且维护简单，很常用。

缓存更新不及时问题

临时缓存是有 TTL 的，如果 60 秒内修改了用户的昵称，缓存是不会马上更新的。最糟糕的情况是在 60 秒后才会刷新这个用户的昵称缓存，显然这会给系统带来一些不必要的麻烦。其实对于这种缓存数据刷新，可以分成几种情况，不同情况的刷新方式有所不同，接下来我给你分别讲讲。

1. 单条实体数据缓存刷新

单条实体数据缓存更新是最简单的一个方式，比如我们缓存了 9527 这个用户的 info 信息，当我们对这条数据做了修改，我们就可以在数据更新时同步更新对应的数据缓存：

复制代码

```
Type UserInfo struct {
  Id         int    `gorm:"column:id;type:int(11);primary_key;AUTO_INCREMENT" j
  Uid        int    `gorm:"column:uid;type:int(4);NOT NULL" json:"uid"`
  NickName   string `gorm:"column:nickname;type:varchar(32) unsigned;NOT NULL"
  Status     int16  `gorm:"column:status;type:tinyint(4);default:1;NOT NULL" js
  CreateTime int64  `gorm:"column:create_time;type:bigint(11);NOT NULL" json:"c
  UpdateTime int64  `gorm:"column:update_time;type:bigint(11);NOT NULL" json:"u
}

//更新用户昵称
func (m *UserInfo)UpdateUserNickname(ctx context.Context, name string, uid int)
  //先更新数据库
  ret, err := m.db.UpdateUserNickNameById(ctx, uid, name)
  if ret {
        //然后清理缓存，让下次读取时刷新缓存，防止并发修改导致临时数据进入缓存
        //这个方式刷新较快，使用很方便，维护成本低
    Redis.Del("user_info_" + strconv.Itoa(uid))
  }
  return ret, count, err
}

```

整体来讲就是先识别出被修改数据的 ID，然后根据 ID 删除被修改的数据缓存，等下次请求到来时，再把最新的数据更新到缓存中，这样就会有效减少并发操作把脏数据带入缓存的可能性。

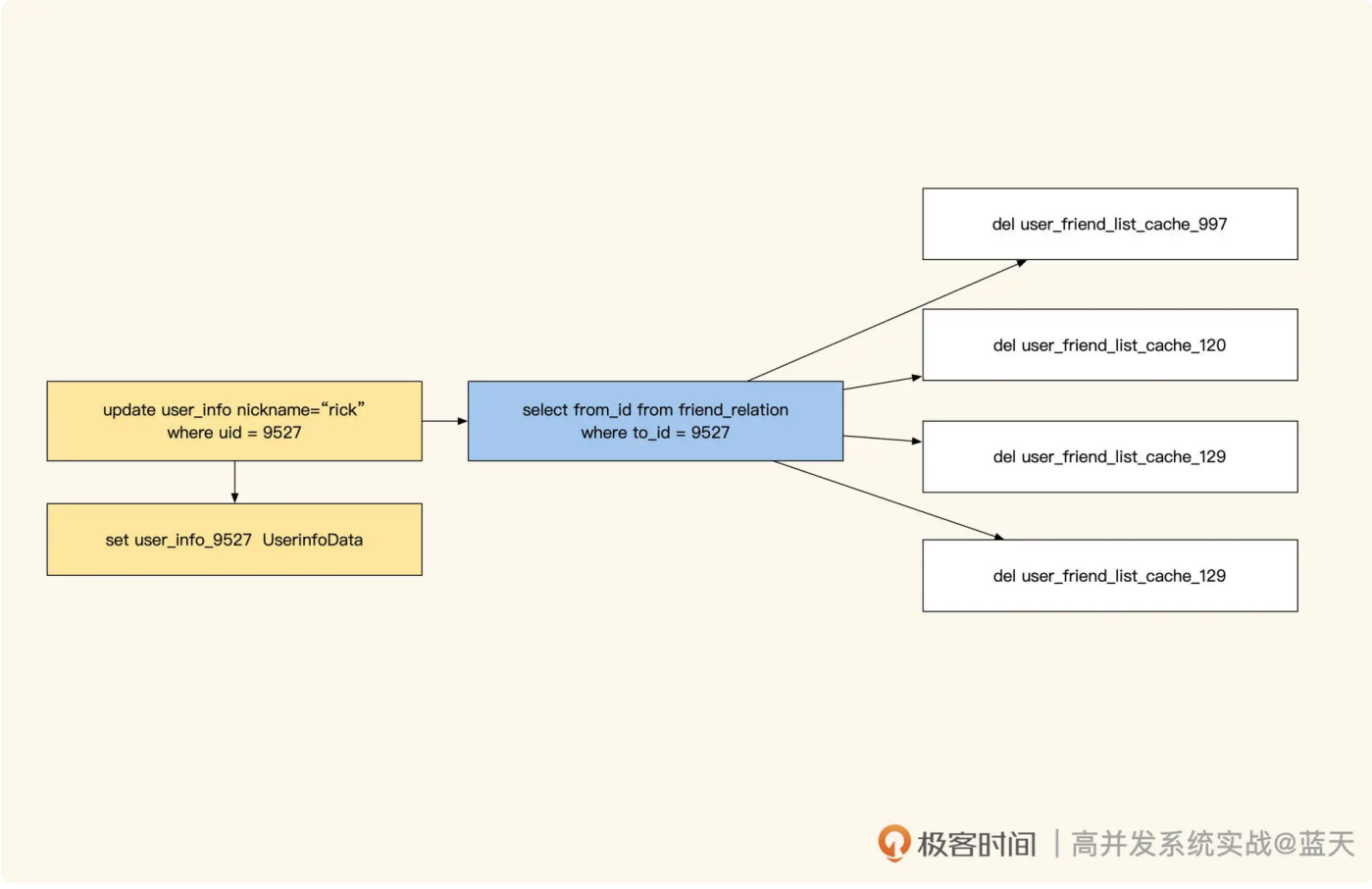
除此之外，我们也可以给队列发更新消息让子系统更新，还可以开发中间件把数据操作发给子系统，自行决定更新的数据范围。

不过，通过队列更新消息这一步，我们还会碰到一个问题——条件批量更新的操作无法知道具体有多少个 ID 可能有修改，常见的做法是：先用同样的条件把所有涉及的 ID 都取出来，然后 update，这时用所有相关 ID 更新具体缓存即可。

2. 关系型和统计型数据缓存刷新

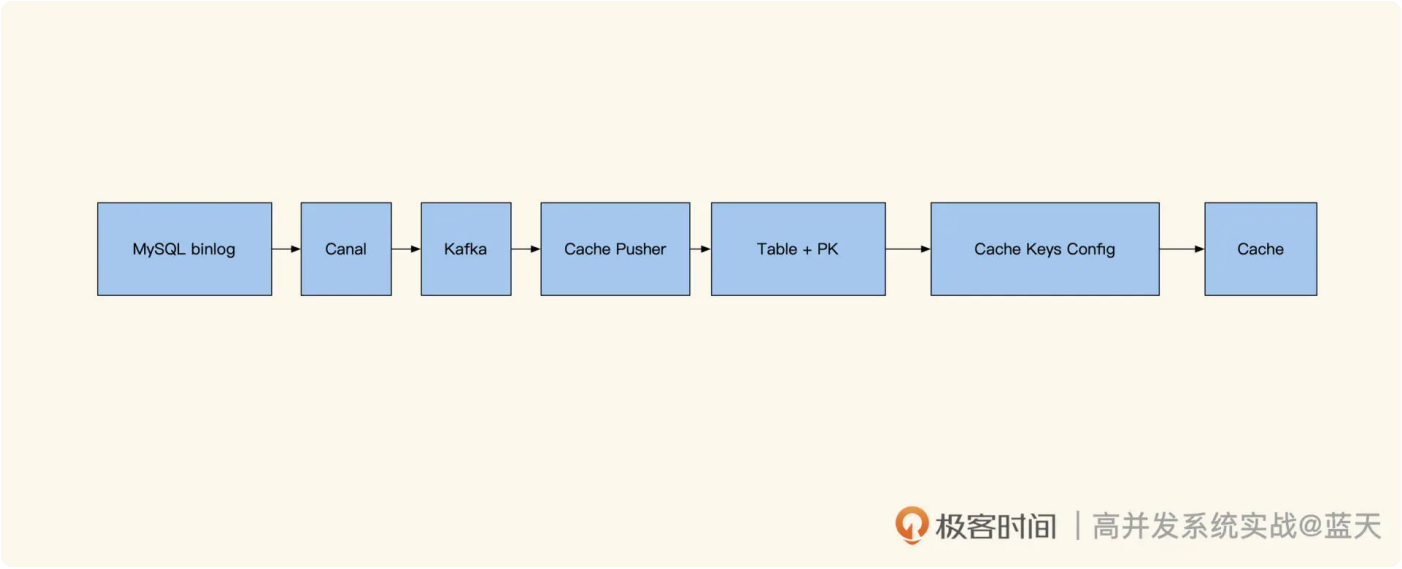
关系型或统计型缓存刷新有很多种方法，这里我给你讲一些最常用的。

首先是**人工维护缓存方式**。我们知道，关系型数据或统计结果缓存刷新存在一定难度，核心在于这些统计是由多条数据计算而成的。当我们对这类数据更新缓存时，很难识别出需要刷新哪些**关联**缓存。对此，我们需要人工在一个地方记录或者定义特殊刷新逻辑来实现相关缓存的更新。



不过这种方式比较精细，**如果刷新缓存很多，那么缓存更新会比较慢，并且存在延迟**。而且人工书写还需要考虑如何查找到新增数据关联的所有 ID，因为新增数据没有登记在 ID 内，人工编码维护会很麻烦。

除了人工维护缓存外，还有一种方式就是通过**订阅数据库来找到 ID 数据变化。**如下图，我们可以使用 Maxwell 或 Canal，对 MySQL 的更新进行监控。



通过记录ID关系刷新缓存

这样变更信息会推送到 Kafka 内，我们可以根据对应的表和具体的 SQL 确认更新涉及的数据 ID，然后根据脚本内设定好的逻辑对相 关 key 进行更新。例如用户更新了昵称，那么缓存更新服务就能知道需要更新 user_info_9527 这个缓存，同时根据配置找到并且删除其他所有相关的缓存。

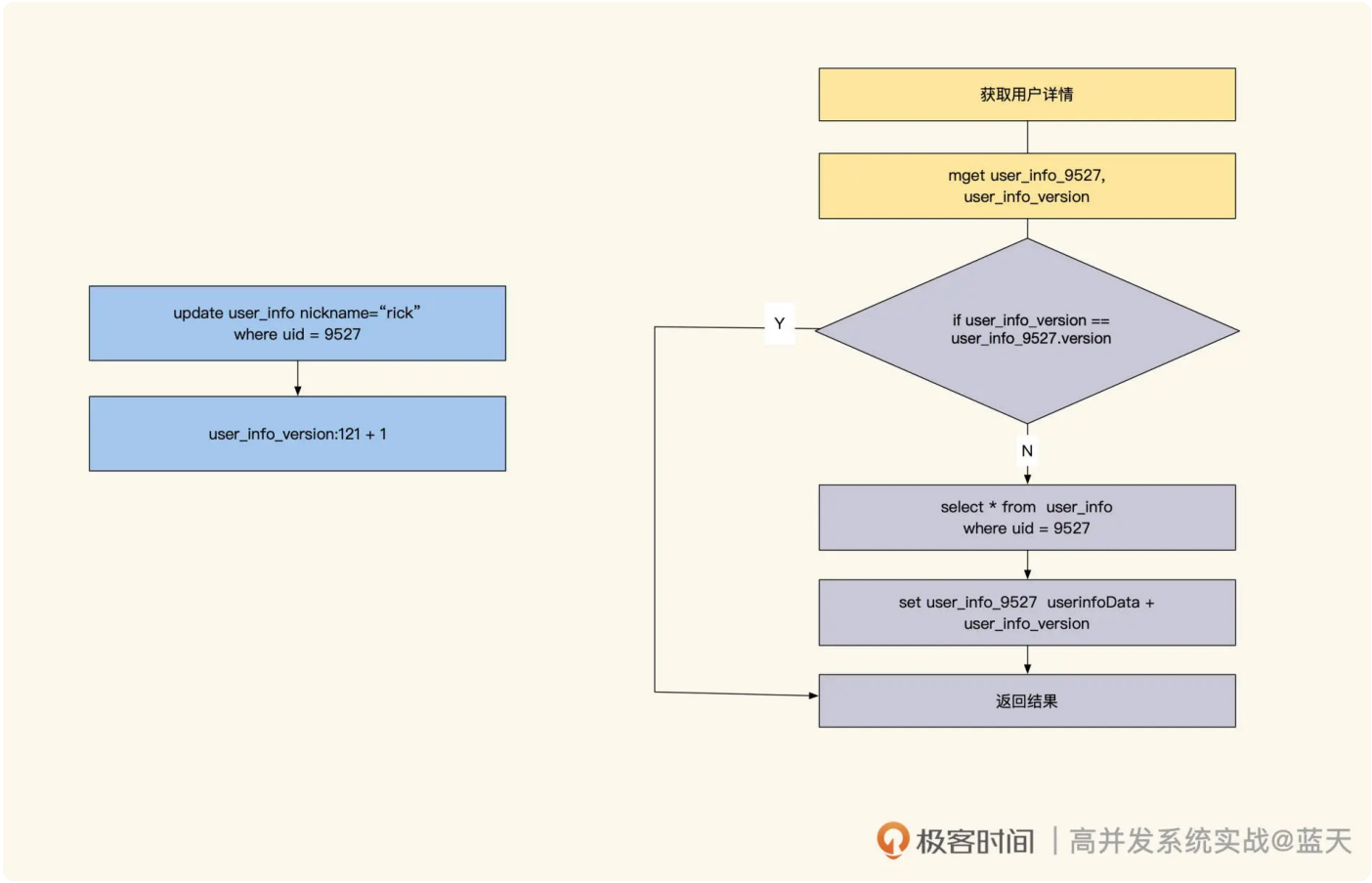
很显然，这种方式的好处是能及时更新简单的缓存，同时核心系统会给子系统广播同步数据更改，代码也不复杂；缺点是复杂的关联关系刷新，仍旧需要通过人工写逻辑来实现。

如果我们表内的数据更新很少，那么可以采用**版本号缓存设计**。

这个方式比较狂放：一旦有任何更新，整个表内所有数据缓存一起过期。比如对 user_info 表设置一个 key，假设是 user_info_version，当我们更新这个表数据时，直接对 user_info_version 进行 incr +1。而在写入缓存时，同时会在缓存数据中记录 user_info_version 的当前值。

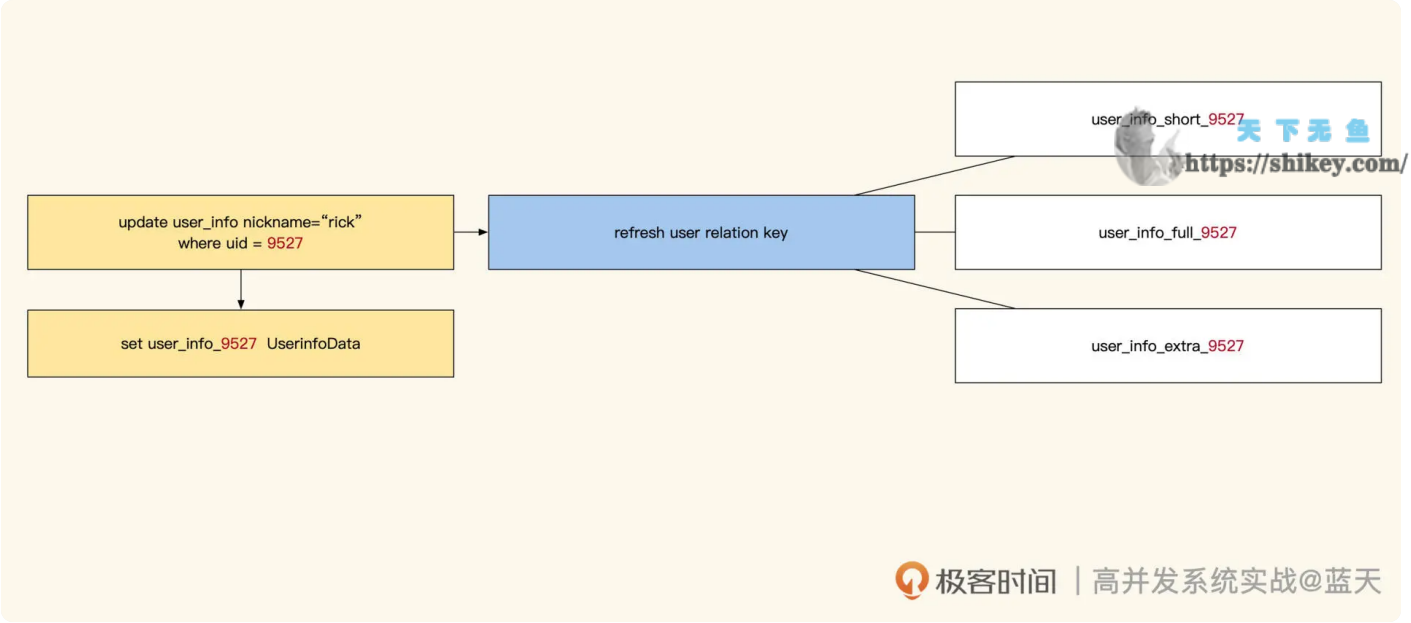
当业务要读取 user_info 某个用户的信息的时候，业务会同时获取当前表的 version。如果发现缓存数据内的版本和当前表的版本不一致，那么就会更新这条数据。但如果 version 更新很频繁，就会严重降低缓存命中率，所以这种方案适合更新很少的表。

当然，我们还可以对这个表做一个范围拆分，比如按 ID 范围分块拆分出多个 version，通过这样的方式来减少缓存刷新的范围和频率。



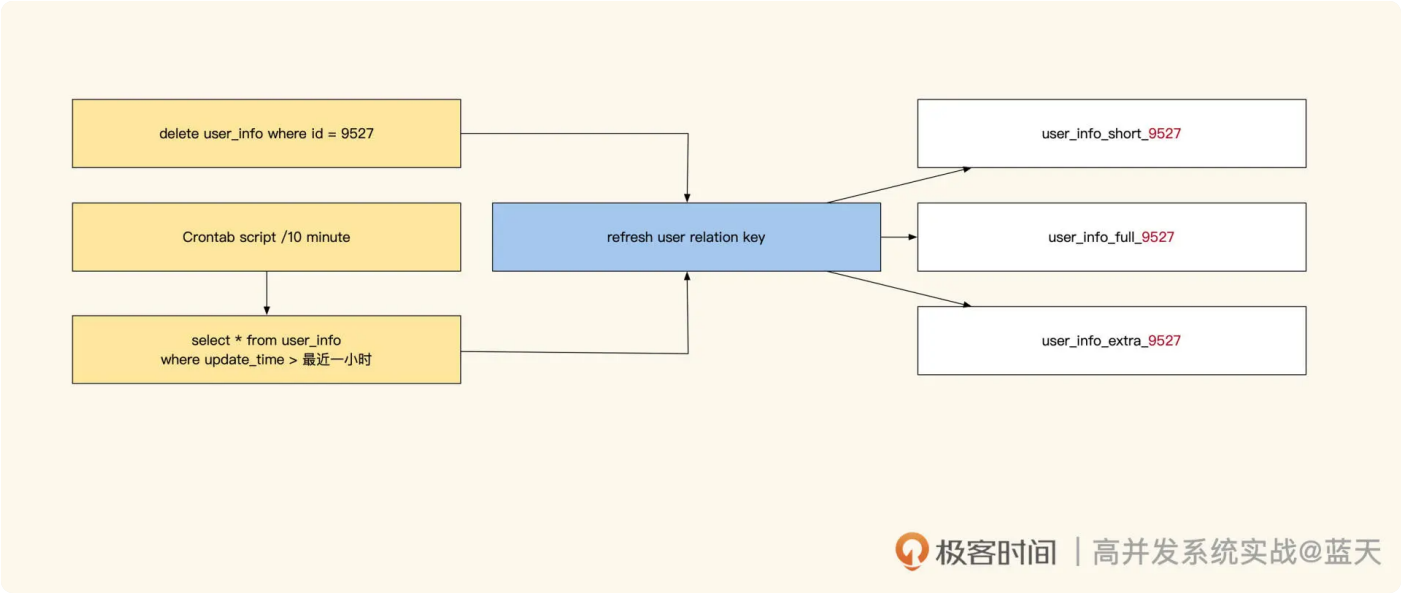
版本号方式刷新缓存

此外，关联型数据更新还可以通过识别**主要实体 ID** 来刷新缓存。这要保证其他缓存保存的 key 也是主要实体 ID，这样当某一条关联数据发生变化时，就可以根据主要实体 ID 对所有缓存进行刷新。这个方式的缺点是，我们的缓存要能够根据修改的数据反向找到它关联的主体 ID 才行。



通过主要实体ID刷新缓存

最后，我再给你介绍一种方式：**异步脚本遍历数据库刷新所有相关缓存**。这个方式适用于两个系统之间同步数据，能够减少系统间的接口交互；缺点是删除数据后，还需要人工删除对应的缓存，所以更新会有延迟。但如果能配合订阅更新消息广播的话，可以做到准同步。



遍历数据库刷新缓存

长期热数据缓存

到这里，我们再回过头看看之前的临时缓存伪代码，它虽然能解决大部分问题，但是请你想一想，当 TTL 到期时，**如果大量缓存请求没有命中，透传的流量会不会打沉我们的数据库？**这其实就是行业里常提到的缓存穿透问题，如果缓存出现大规模并发穿透，那么很有可能导致我们服务宕机。

所以，数据库要是扛不住平时的流量，我们就不能使用**临时缓存**的方式去设计缓存系统，只能用**长期缓存**这种方式来实现**热点缓存**，以此避免缓存穿透打沉数据库的问题。不过，要想实现长期缓存，就需要我们人工做更多的事情来保持缓存和数据表数据的一致性。

要知道，长期缓存这个方式自 NoSQL 兴起后才得以普及使用，主要原因在于长期缓存的实现和临时缓存有所不同，它要求我们的业务**几乎完全不走数据库**，并且服务运转期间所需的数据都要能在缓存中找到，同时还要保证使用期间缓存不会丢失。

由此带来的问题就是，我们需要知道缓存中具体有哪些数据，然后提前对这些数据进行预热。当然，如果数据规模较小，那我们可以考虑把全量数据都缓存起来，这样会相对简单一些。

为了加深理解，同时展示特殊技巧，下面我们来看一种“临时缓存 + 长期热缓存”的一个有趣的实现，这种方式会有小规模缓存穿透，并且代码相对复杂，不过总体来说成本是比较低的：

复制代码

```
// 尝试从缓存中直接获取用户信息
userinfo, err := Redis.Get("user_info_9527")
if err != nil {
  return nil, err
}

//缓存命中找到，直接返回用户信息
if userinfo != nil {
  return userinfo, nil
}

//set 检测当前是否是热数据
//之所以没有使用Bloom Filter是因为有概率碰撞不准
//如果key数量超过千个，建议还是用Bloom Filter
//这个判断也可以放在业务逻辑代码中，用配置同步做
isHotKey, err := Redis.SISMEMBER("hot_key", "user_info_9527")
if err != nil {
  return nil, err
}

//如果是热key
if isHotKey {
  //没有找到就认为数据不存在
  //可能是被删除了
  return "", nil
}

//没有命中缓存，并且没被标注是热点，被认为是临时缓存，那么从数据库中获取
//设置更新锁set user_info_9527_lock nx ex 5
//防止多个线程同时并发查询数据库导致数据库压力过大
```

```
lock, err := Redis.Set("user_info_9527_lock", "1", "nx", 5)
if !lock {
  //没抢到锁的直接等待1秒 然后再拿一次结果，类似singleflight实现
  //行业常见缓存服务，读并发能力很强，但写并发能力并不好
  //过高的并行刷新会刷沉缓存
  time.sleep( time.second)
  //等1秒后拿数据，这个数据是抢到锁的请求填入的
  //通过这个方式降低数据库压力
  userinfo, err := Redis.Get("user_info_9527")
  if err != nil {
    return nil, err
  }
  return userinfo,nil
}

//拿到锁的查数据库，然后填入缓存
userinfo, err := userInfoModel.GetUserInfoById(9527)
if err != nil {
  return nil, err
}

//查找到用户信息
if userinfo != nil {
  //将用户信息缓存，并设置TTL超时时间让其60秒后失效
  Redis.Set("user_info_9527", userinfo, 60)
  return userinfo, nil
}

// 没有找到，放一个空数据进去，短期内不再问数据库
Redis.Set("user_info_9527", "", 30)
return nil, nil
```


可以看到，这种方式是长期缓存和临时缓存的混用。当我们要查询某个用户信息时，如果缓存中没有数据，长期缓存会直接返回没有找到，临时缓存则直接走更新流程。此外，我们的用户信息如果属于热点 key，并且在缓存中找不到的话，就直接返回数据不存在。

在更新期间，为了防止高并发查询打沉数据库，我们将更新流程做了简单的 singleflight（请求合并）优化，只有先抢到缓存更新锁的线程，才能进入后端读取数据库并将结果填写到缓存中。而没有抢到更新锁的线程先 sleep 1 秒，然后直接读取缓存返回结果。这样可以保证后端不会有多个线程读取同一条数据，从而冲垮缓存和数据库服务（缓存的写并发没有读性能那么好）。

另外，hot_key 列表（也就是长期缓存的热点 key 列表）会在多个 Redis 中复制保存，如果要读取它，随机找一个分片就可以拿到全量配置。

这些热缓存 key，来自于统计一段时间内数据访问流量，计算得出的热点数据。那长期缓存的更新会异步脚本去定期扫描热缓存列表，通过这个方式来主动推送缓存，同时把 TTL 设置成更长的时间，来保证新的热数据缓存不会过期。当这个 key 的热度过去后，热缓存 key 就会从当前 set 中移除，腾出空间给其他地方使用。

当然，如果我们拥有一个很大的缓存集群，并且我们的数据都属于热数据，那么我们大可以脱离数据库，将数据都放到缓存当中直接对外服务，这样我们将获得更好的吞吐和并发。

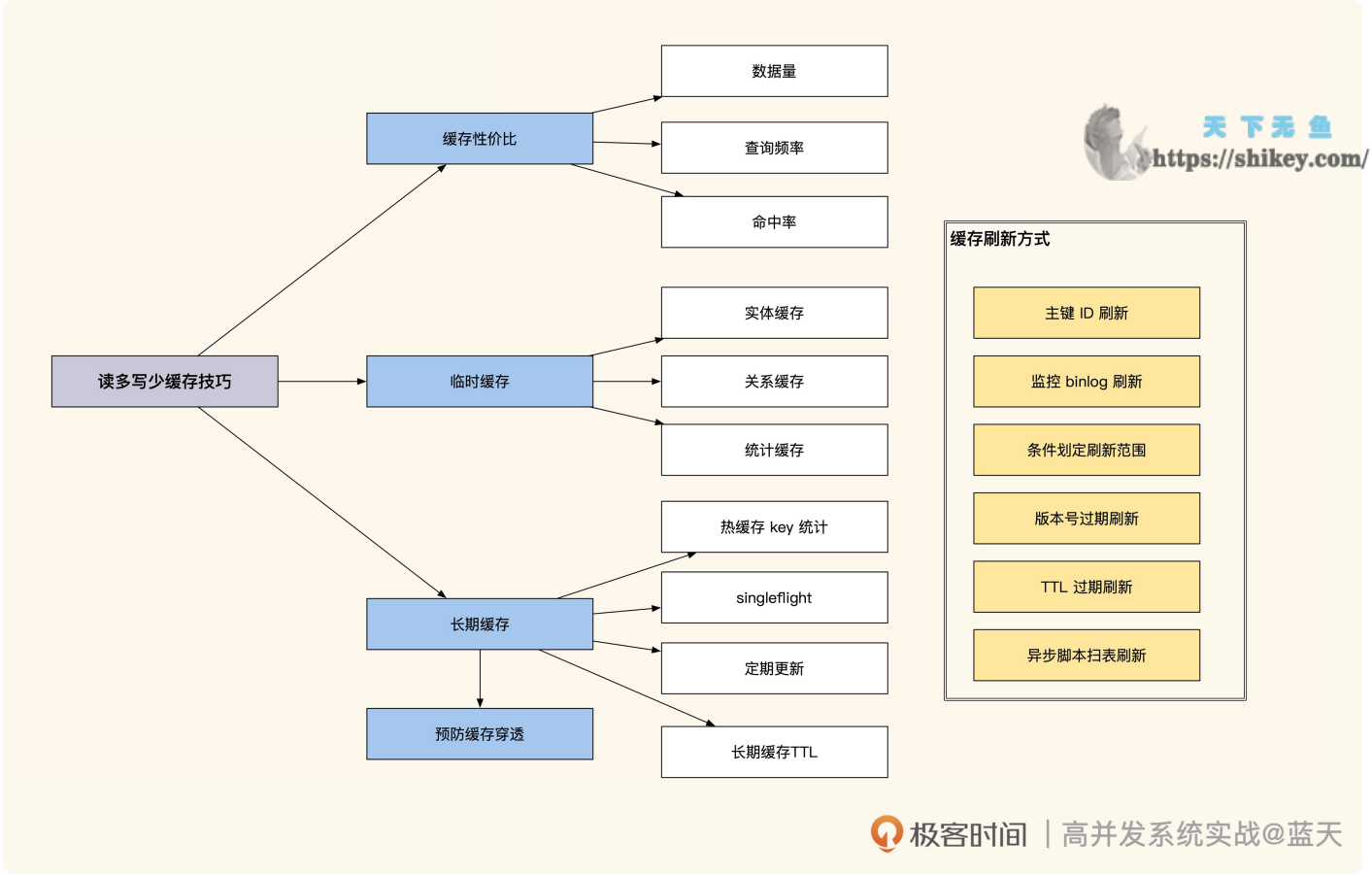
最后，还有一种方式来缓解热点高并发查询，在每个业务服务器上部署一个小容量的 Redis 来保存热点缓存数据，通过脚本将热点数据同步到每个服务器的小 Redis 上，每次查询数据之前都会在本地小 Redis 查找一下，如果找不到再去大缓存内查询，通过这个方式缓解缓存的读取性能。

总结

通过这节课，我希望你能明白：不是所有的数据放在缓存就能有很好的收益，我们要从**数据量**、**使用频率**、**缓存命中率**三个角度去分析。读多写少的数据做缓存虽然能降低数据层的压力，但要根据一致性需求对其缓存的数据做更新。其中，单条实体数据最容易实现缓存更新，但是有条件查询的统计结果并不容易做到实时更新。

除此之外，如果数据库承受不了透传流量压力，我们需要将一些热点数据做成**长期缓存**，来防止大量请求穿透缓存，这样会影响我们的服务稳定。同时通过 singleflight 方式预防临时缓存被大量请求穿透，以防热点数据在从临时缓存切换成热点之前，击穿缓存，导致数据库崩溃。

读多写少的缓存技巧我还画了一张导图，如下所示：



思考题

1. 使用 Bloom Filter 识别热点 key 时，有时会识别失误，进而导致数据没有找到，那么如何避免这种情况呢？

2. 使用 Bloom Filter 只能添加新 key，不能删除某一个 key，如果想更好地更新维护，有什么其他方式吗？

欢迎你在留言区与我交流讨论，我们下节课见！

上一篇 01 | 结构梳理：大并发下，你的数据库表可能成为性能瓶颈
下一篇 03 | Token：如何降低用户身份鉴权的流量压力？

精选留言 (12)

写留言

peter 置顶

2022-10-26 来自北京

请教老师几个问题：

Q1：缓存都是有超时时间的，从这个意义上说，都是“临时”的，为什么本文还要分为“临时”缓存和“长期”缓存？

Q2：“临时”缓存和“长期”缓存在实现上可以用同一个软件吗？

比如，两者都可以用Redis实现？或者，“临时缓存”是用一个组件实现（非Redis）而“长期缓存”用Redis实现？ 或者，“临时缓存”在代码中实现而“长期缓存”用Redis？

Q3：人工维护缓存，怎么操作？

缓存数据一般都比较多，人工怎么能够维护一堆数据？具体是怎么操作的？ 有一个界面，通过此界面来操作吗？

作者回复: 你好，peter，又见面了，感谢你的留言～

Q1:由于我们的大多数数据都是有时效性的，我们很少去做永久的内存缓存，毕竟内存还是很贵的，我们需要考虑性价比。长期缓存可以是一天，临时TTL是30秒。同时长期的更新是定期脚本刷新，临时是用到才会放进去一会儿。再来看，长期的访问很频繁，如果放开会导致数据库压力很大，但是临时的由于访问量小所以不用特意防击穿。

Q2:如果缓存压力不大可以用一个，如果很大会再做个L1缓存，在每台业务服务器上，这样能缓解核心缓存压力。

Q3：这个属于人工写代码，如我更新了用户的昵称，那么我会刷新这个用户的所有帖子的缓存，以及这个用户的所有最近留言缓存，以及这个用户的个人信息缓存。这是一种，还有一种就是你说的界面配置规则，但是这些都是要能快速定位的才可以。你可能会碰到，用户昵称以rick开头的账号有多少个，当我们改昵称为这个的时候，对于这种条件多样的，刷新哪个缓存不好确定，只能临时缓存30秒等他过期后刷新

3

Daniel 置顶

2022-10-27 来自北京

1. 使用 Bloom Filter 识别热点 key 时，有时会识别失误，进而导致数据没有找到，那么如何避免这种情况呢？

通过我的 “机器学习的经验”，我觉得是这个布隆过滤器的哈希算法有点过拟合了，也就是说容错率高了，在资金充足的情况下先试着调低“容错率“(超参数)提升容量试试（不知道工业界

上布隆过滤器的容错率能设置成 0%吗？但后期可能随着数据量的增长也是一个无限扩容的”吞金兽“呀），但我感觉我这个想法在工业界应该不成立。

第二种方法，我想到的是，如果这个 key 被误识别为”hotkey”的话，就在内存中记入“not_hot key”列表，每次数据进来的时候，先用 缓存里的 not_hotkey里的列表来筛，要是不是hotkey就做成临时缓存，要是这个key是hotkey的话，就进行长期缓存来处理。

2. 使用 Bloom Filter 只能添加新 key，不能删除某一个 key，如果想更好地更新维护，有什么其他方式吗？

对于长时间不用的 key ，我认为可以设置一个“失效时间”，比如 一周内不用，就自动清除掉这个key。
之后在新的一周，把失效的key清理出去，再重新整理好一个列表，重新更新一遍这个布隆过滤器的新的哈希算法表。
（但感觉这个方法貌似不是最优的，也要在半夜用户量访问少的时间点去做变更处理）

老师想请教一个问题，对于 hotkey （热点数据）这个工业界的评价标准是不是不同行业会不一样呀？
比较想知道工业界上是用什么方法（一般统计方法？机器学习聚类？深度学习网络？）和工具（数据埋点？用户操作行为分析？），来做 “热点数据”的 判别的？

作者回复: 你好，Daniel，很高兴收到你的留言

第一种方法，如果使用集合检测也可以达到同样效果，你提到的方法缺陷在于not_hotkey会很大，这里需要记录所有不是hot key的所有key。目前bloomfilter只是个模糊筛选，用小量数据换更好的性能，但是他确实误判概率多一些。

第二个问题也是一个办法，但是需要我们能够精准控制这一批数据过期时间，但是我们在这节课用它主要是为了判断本地缓存中是否有这个缓存，由于无法判断所以需要每次询问，会导致系统更加复杂。

第三个问题 主要是访问量，我们可以将一些key 做一些count统计，数据埋点就足够了，机器学习和深度学习的QPS有些低，我印象里，一个1w元的显卡做发音评分，一秒钟只能处理4个请求，场景不太适合，有点小才大用了。

共 3 条评论 >

2

Elvis Lee
2022-10-26 来自北京

1. 使用 Bloom Filter 识别热点 key 时，有时会识别失误，进而导致数据没有找到，那么如何避免这种情况呢？
布隆可以判断一定不存在的数据，那么是否可以认为，只要插入不成功，即为热数据，但在设计布隆的时候需要根据业务来设置好容量和容错率。同时布隆删除操作在生产上不建议，最好是持久化后用版本号去区分。如果是离线链路，更推荐生成布隆文件，推送去客户端。
实时的,目前接触是保存在Redis,Redis7的版本好像已经不需要插件

> 作者回复: 你好，Elvis Lee，很高兴收到你的留言，不建议这样使用bloomfilter，主要原因在于，他的识别和md5计算结果一样，有的时候不同的key返回的结果是一致的，所以这样是拿不到准确结果的。

共 4 条评论 >　　2

传输助手
2022-10-26 来自北京

读取数据库设置缓存的时候，为了不受数据库主从延迟的影响，是不是需要强制读主库？

> 作者回复: 你好，传输助手，很高兴收到你的留言，这里有一个前提，就是我们加缓存的服务基本都是读并发高的服务，对于MySQL主库来说，问题就是全局只有一个主库，所以他是单点，同时更脆弱，理论上这种读压力尽量不要压到主库上～

3

一步 酥
2022-11-03 来自北京

1. Bloom Filter 存在误报，会把不是热点的 key 识别成热点key, 所以需要一个 0误报的算法，数据结构 所以 Cuckoo Filter 布谷鸟过滤器来了
2. 可以定期或者其他策略 重新构造 Bloom Filter

> 其实上面的2个问题，都可以使用 Cuckoo Filter 来解决

> 作者回复: 你好，很高兴收到你的回复，没错！确实Cuckoo Filter能够解决所有问题！同时补充提醒：他也有一些缺点使用的时候要注意，性能没有bf高，同时删除存在误删情况～

共 2 条评论 >　　1

hhh
2022-11-11 来自北京

没有抢到锁的sleep 1s然后去查询，这样接口耗时不是就会肯定大于1s吗 ，假如超时配置小于1s，这次请求不是必定会超时嘛

作者回复: 你好，hhh，时间不是绝对的这个可以根据情况进行调整，以前我们普遍是200ms左右，相对的这个方式比直接打沉中心数据库好那么一点，前提我们前端服务器足够多。

共 3 条评论>

Geek_lucas

2022-11-09 来自北京

1、key-->hotkey布隆-->不存在-->非hotkey-->进入非hotkey布隆

2、key-->hotkey布隆-->存在-->可能是hotkey-->非hotkey布隆-->不存在-->可能是hotkey，当做hotkey处理

3、key-->hotkey布隆-->存在-->可能是hotkey-->非hotkey布隆-->存在-->可能是非hotkey，当做非hotkey处理

作者回复: 你好，是说另外记录一个key来记录他是否是hot_key吗

Geek_lucas

2022-11-09 来自北京

1. 使用 Bloom Filter 识别热点 key 时，有时会识别失误，进而导致数据没有找到，那么如何避免这种情况呢？

答：学习数据结构的时候，我记得布隆过滤器，如果判断一个key命中，那么他【可能】有，如果判断一个key没有命中，那么他【一定】没有。所以应该反着来用，就是用来判断一个key，如果在hotkey布隆过滤器中没有命中，它一定不是hotkey。

2. 使用 Bloom Filter 只能添加新 key，不能删除某一个 key，如果想更好地更新维护，有什么其他方式吗？

无

作者回复: 你好，第一种有缺陷，因为我们的数据集很大，比如亿的key，第二种其实有办法只是代价选择

SunshineBoy

2022-11-01 来自北京

哈喽 大佬 redis适合做app中一级、二级页面降级方案的存储吗？如果存储的value比较大，有没有推荐的降级方案？

作者回复: 你好，建议对象存储配合cdn缓存实现

共 3 条评论>

Sky
2022-10-29 来自北京

在更新期间，为了防止高并发查询打沉数据库，我们将更新流程做了简单的 singleflight（请求合并）优化，只有先抢到缓存更新锁的线程，才能进入后端读取数据库并将结果填写到缓存中。而没有抢到更新锁的线程先 sleep 1 秒，然后直接读取缓存返回结果。这样可以保证后端不会有多个线程读取同一条数据，从而冲垮缓存和数据库服务（缓存的写并发没有读性能那么好）。

并发更新的时候，为了防止超卖等问题，是不是最好还要在sql中加上乐观锁CAS？

作者回复: 你好，sky，很高兴收到你的留言，这样数据库压力会更大，这时需要特殊做，后续会讲到强一致怎么做

不吃包子
2022-10-28 来自北京

针对1.2的问题，
搜索到了如下解决方案： 调整布隆过滤器参数或者用布谷鸟过滤器。
我想说说我自己的看法，针对误判的情况，能不能再加一层缓存？比如说一个数据被误判为有，则去查询数据库了，这个时候为空，记到缓存里面，如果下次再访问该数据的时候，直接从缓存返回。针对问题2 同样也维护一个删除的缓存。

作者回复: 你好，不吃包子，很高兴收到你的留言，布谷鸟过滤器是一个不错的解决方案，另外查找不到缓存空也不错～

共 2 条评论>

门窗小二
2022-10-28 来自北京

老师！但是课后题有答疑篇吗？

编辑回复: 建议先自己尝试回答，课后题答案后续看回答情况再公布。