(八)漫谈分布式之缓存篇：唠唠老生常谈的MySQL与Redis数据一致性问题！

==========

> 本文为稀土掘金技术社区首发签约文章，30天内禁止转载，30天后未获授权禁止转载，侵权必究！

引言

--

从开设《漫谈分布式专栏》至今，已经写了许多与一致性相关的文章，可其中大部分并不能和日常开发工作挂钩，为此，本文来聊一个跟实际工作挂钩的老生常谈的问题：**分布式系统中的缓存一致性**。

缓存技术，既能减轻数据库访问压力，又能加快请求响应速度，这是一件两全其美的事情，所以稍具规模的系统，都会引入缓存机制来达成这两个目的。缓存技术从最初的本地缓存发展到如今，已经走进以`Redis`为代表的分布式缓存时代，尤其是现在的分布式系统，`Redis`这类中间件已成为系统不可或缺的基础设施。

之前说过，任何技术有利必有弊，我们不能光盯着优势看，对于劣势同样要谨记于心，分布式缓存中最经典的就是缓存一致性问题，工作中经常接触，也是各种大大小小面试中，分布式领域里被问频率最高的问题之一。也正因为面试问的多，所以围绕着“缓存一致性”这个话题出现很多八股文，其中不乏有些乍一看令人拍手称赞的奇淫技巧，可经过仔细一推敲就会发现，好像行不通、有点华而不实……。好了，简单交代本文核心后，下面来好好唠唠缓存一致性这个问题~

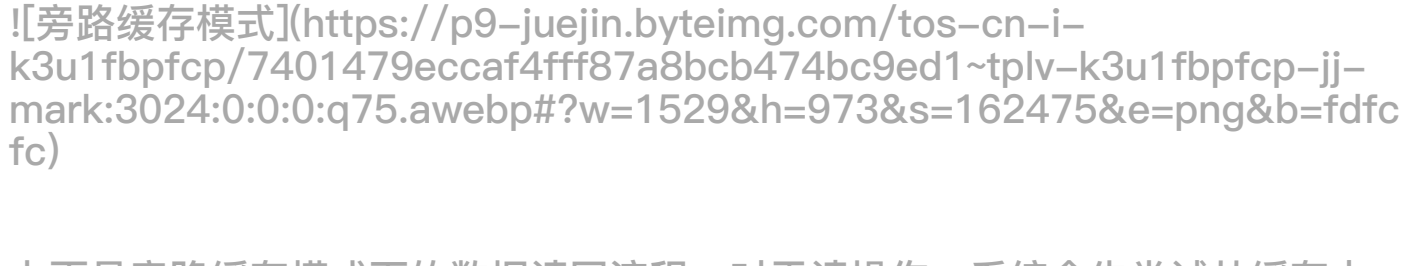
> PS：个人编写的[《技术人求职指南》](http://cxyroad.com/”https://s.juejin.cn/ds/USoa2R3/”)小册已完结，其中从技术总结开始，到制定期望、技术突击、简历优化、面试准备、面试技巧、谈薪技巧、面试复盘、选`Offer`方法、新人入职、进阶提升、职业规划、技术管理、涨薪跳槽、仲裁赔偿、副业兼职……，为大家打造了一套“从求职到跳槽”的一条龙服务，同时也为诸位准备了七折优惠码：`3DoleNaE`，近期需要找工作的小伙伴可以点击：[s.juejin.cn/ds/USoa2R3/](http://cxyroad.com/”https://s.juejin.cn/ds/USoa2R3/”)了解详情！

一、常见的缓存设计模式

根据系统的业务需求、性能需求、一致性要求等方面，在日常开发中，不同的系统可能会采用不同的缓存设计模式。在不同的业务场景下，选用合适的缓存设计模式或许能带来更好的性能收益，可究竟怎么根据业务特性来选用合适的模式呢？我们先来看看几种常见的缓存设计模式。

1.1、旁路缓存模式

旁路缓存模式（`Cache-Aside Pattern`）是应用范围最广泛的缓存设计模式，使用该模式的应用程序，会同时操作缓存与数据库，如下：



上面是旁路缓存模式下的数据读写流程，对于读操作，系统会先尝试从缓存中获取数据，如果命中缓存，则直接返回缓存数据；如果未命中，就会去数据库里查询数据，而后再更新缓存，最后返回`DB`中查到的数据。对于写操作，系统先更新数据库，接着再主动删除对应的缓存，使之前加载过的缓存失效。

上面说的这种写入方式被叫做`Write-Around`，即直接写数据库不写缓存，什么时候写缓存呢？在当前数据被更新后，因为会主动删除之前的缓存，所以在下次出现对应的读取请求时，无法命中对应的缓存，就会从数据库查出最新的数据并加载进缓存中。

1.2、读写穿透模式

读写穿透模式，其实是读穿透（`Read-Through Pattern`）、写穿透（`Write-Through Pattern`）的组合体。

读穿透模式，是指处理读操作时，应用程序只会和缓存层进行交互，如果缓存层中没有的数据，再由缓存层负责与数据库进行交互，示意图如下：

mark:3024:0:0:0:q75.awebp#?w=1117&h=908&s=118339&e=png&b=fdfcfc)

实际读的流程和旁路缓存很类似，只不过在读穿透模式中，抽象出了一个缓存层，业务系统只需要跟缓存层进行交互，不管数据在缓存、还是在数据库里，最终都会由缓存层向业务系统返回数据。不过为了减少缓存层查库的次数，读穿透一般会结合写穿透一起使用。

写穿透模式，即是指应用程序在处理读操作时，会先更新缓存、再数据库，两个动作放在一个事务里，从而保证同时成功：

![写穿透](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/eeab5c4f0f0e4a49aa127d8b5d9be044~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=817&h=896&s=86587&e=png&b=fdfcfc)

如果单看这种写穿透模式，其实特别鸡肋，一方面将写缓存、写库绑定在一起，增加了业务系统处理写操作的风险；另一方面，先写缓存再写库，还带来了额外的写延迟问题。不过结合读穿透一起使用，因为读操作会先走缓存层，所以写库的延迟可以不用关心~

综合来看，读写穿透模式还是存在较大的劣势，因为系统需要抽象出一个缓存层，所有读写操作都放到缓存层去实现，而业务层则调用对应的实现即可，这显然增加了开发的工作量。除此之外，对应一些较为复杂的缓存数据，例如从多张表里聚合出来的信息，在这种模式下处理起来就会很麻烦。

1.3、异步写入模式

异步写入模式（`Write-Behind Pattern`），也被称为写后模式，它跟前面的写穿透模式有点类似，都是先写缓存，不过与之不同的是：**异步写入模式中，只要缓存更新成功后，就会给客户端返回写操作处理成功，而对于写库的动作，则是异步去完成**。

![异步写入](https://p1-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/3dfa5030267e4ada9a3515067d98e40e~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=819&h=841&s=84307&e=png&b=fcfafa)

这种模式和`MQ`做削峰有着异曲同工之妙，因为大多关系型数据库都是基于磁盘工作，因此会导致写数据时性能一般，为了获得更高的写入，数据会先写入到缓存中，然后记录到等待写库的列表里（例如`Key`的集合），**当待写入的

数据条数达到一定量级后，才会触发批量写库动作将数据落库**。

上面这种模式，显然能让系统具备更高的写入能力，**毕竟将分散性的多次写入，合并成了一次批量写入**。在高并发场景下，这种方案用的不算少，比如秒杀活动，提前将商品库存拆成十份放进`Redis`，进一步精细化锁的颗粒度，每隔一段时间，再求和`Redis`中的剩余库存，并将最新的库存数刷到数据库。

不过这种模式虽然能获得更高的写入能力，但也会带来更多的问题，因为缓存的数据在内存里，所以存在丢失风险；又或者做异步批量写入时，如果其中某一条数据写入失败怎么处理？又好比异步写入时，如何保证事务的一致性……，这都是在使用这种方案时要考虑的问题。

还有，如果你打算使用这种模式，那一定要先读缓存，不要直接读库，毕竟写库动作被异步化了，缓存中的数据才是最新的，如果冒然从数据库中读取数据，这时可能会读到还未更新的旧数据，从而给系统带来不可逆转的出错风险。

> PS：这种思想在`InnoDB`的`BufferPool`中应用很广泛，而`InnoDB`在读取数据时，都是会先在内存找。

1.4、缓存加载模式

再来聊聊缓存加载模式，所谓的缓存加载，**就是指何时将数据库中的数据加载到缓存中**，总共有两种方式：`Refresh-Ahead Pattern`预刷新模式、`Lazy-Loading Pattern`懒加载模式，后者比较常见，**只有当出现真正的请求获取数据时，才会将数据加入进缓存**，这种方式能节省缓存中间件的内存空间，但会导致首次访问数据时存在一定延迟。

相反，与懒加载对应的就是预加载，**代表提前将可能访问的数据载入到缓存里**，这种方案在实际项目用的也很多，比如电商项目里常见的缓存预热功能，为了避免“首次查询数据库、数据还未载入缓存前”这个窗口里，出现过高的并发流量击穿缓存，一般在大促前就会对可能被访问的热点数据进行进行缓存预加载，这样既能加快首次查询的响应时间，还能避免过高的瞬时并发打垮数据库。

不过想要使用预加载模式，一定要能明确哪些才是真正的热点数据，如果预测不够准确，可能会导致大量内存被白白浪费。预加载模式的另一个劣势，就是会增加工作量，毕竟预热要么通过流量录制+回访来做、要么写脚本或代码完成，无论哪种都需要时间去实现。

二、缓存不一致场景分析

好了，前面讲了几种常用的缓存设计模式，说到底，其实就是缓存载入、数据更新的先后顺序、时间节点不同罢了，下面一起来看看最开始提到的话题：**分布式系统中，如何保证数据库与缓存的一致性**？这里先说结论，**不管是什么方案，都只能满足弱一致性，而无法保证绝对意义上的线性（强）一致性，想要满足真正的强一致性，那就不能用缓存**！

我们来归纳一下系统处理些操作时，对于缓存、数据库可能出现的几种组合：

* ①先写缓存再写库；
* ②先写库再写缓存；
* ③先删缓存再写库；
* ④先写库再删缓存。

上述四种组合，究竟谁更好呢？我们先达成一个共识，即系统处理读操作的过程为“先查询缓存，命中就返回，未命中就查库，如果数据库返回了数据，就把结果集塞进缓存里，最后再返回”。建立在这个基础上，假设目前数据库里有一条数据：

```
+---------+-----------+----------+----------+---------------------+
| user_id | user_name | user_sex | password | register_time       |
+---------+-----------+----------+----------+---------------------+
|       1 | 竹子      | 男       | 6666     | 2024-05-26 04:22:01 |
+---------+-----------+----------+----------+---------------------+
```

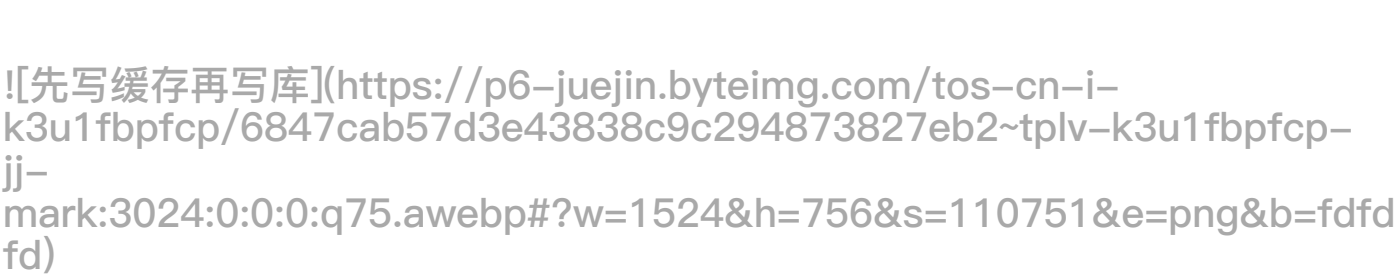
首先出现一个请求读取这条数据，按照前面的说法，这条数据也会被载入到`Redis`里。接着，出现一个更新该数据的请求，对应的`SQL`语句如下：

```
update zz_users set password = ’8888’ where user_id = 1;
```

```

将前面提到的几种组合，套进这个例子里面，下面挨个分析可能会造成不一致的原因。

### 2.1、先写缓存再写库




先些缓存后写库，说明这时会先将`Redis`里的数据，`password`更新成`8888`，接着再更新数据库的数据行，这种方式如果不出现意外，理论上没有问题，毕竟缓存和库都会更新，两者最终的数据肯定是一致的。可意外我们能避免吗？不能，来看一种情况，假设缓存更新成功后，写库的时候报错了，咋办？

因为写库的时候报错，代表这时数据库里数据并未更新，而线程遇到报错后，就会被终止执行，意味着这个请求不会再将数据库的`password`更新为`8888`，此时缓存和数据库之间就出现了不一致性的情况。为此，先更新缓存再更新数据库的这种方案绝对不行，出意外造成不一致的风险太高了。

### 2.2、先写库再写缓存


既然先更新缓存，后面库更新报错就会造成不一致，那么这回先更新数据库，再更新缓存总可以了吧？毕竟数据库有事务呀，如果更新缓存的时候报错，那么数据库对应的变更将会回滚，这样数据库、缓存里的数据都是原来的旧数据，自然没有不一致问题了对不？

如果这么想那就错了，一定不要站在单线程的角度去思考问题时，套进多线程思维，我们先将前面的请求称之为`A`，假设这时又来了一个并发请求`B`，将`ID=1`的`password`更新为`9999`，会有什么问题呢？来看：
```

* ①请求`A`先将数据库更新为`8888`；
* ②请求`B`将数据库更新为`9999`；
* ③请求`B`将缓存更新为`9999`；
* ④请求`A`将缓存更新为`8888`。

大家注意看，可能请求`A`在刚更新完数据库后，网络出现一定波动，导致后来的请求`B`先更新缓存中的值，然后`A`才去更新成`8888`，可是此时库里的数据已经是`B`的`9999`，最终又出现了不一致问题……

> PS：上述场景不一定能出现，要取决于数据库类型，像`MySQL`默认的隔离级别为`RR`，假设这个修改`password`方法上添加了`@Transactional`事务注解，那么`A、B`请求会对应着两个不同的事务，因为`A`先执行、先更新库，而`A、B`操作的又是同一条数据，所以在`A`事务提交之前，`B`肯定会被行锁阻塞，自然不会出现上述问题。

但就算在`RR`级别下，先写库再写缓存就一定能保证一致性吗？答案同样是`No`，比如一个嵌套方法如下：

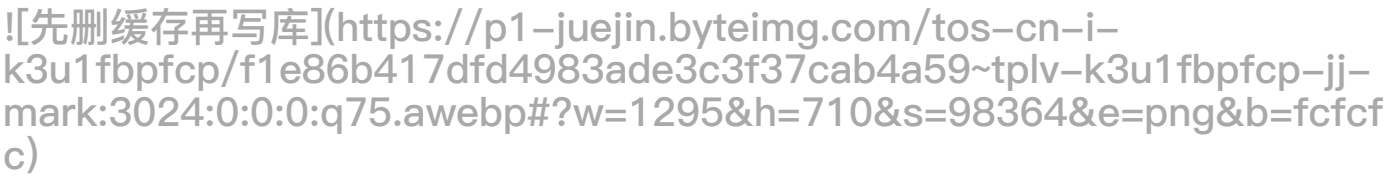
```
@Transactional
public updateMemberInfo(DTO dto) {
    // 1.其他前置逻辑……

    // 2.模拟更新数据库的password值
    updateDbPwd(...);
    // 3.模拟更新缓存中的password值
    updateCachePwd(...);

    // 4.其他后置逻辑……
}
```

正如上述伪代码所示，假设更新`password`仅是`updateMemberInfo()`方法的一小部分逻辑，而此方法执行到伪代码中的第四处时报错，因为方法上有事务注解，这时库中更改的`password`值会被回滚，而缓存中的并不会，毕竟`Redis`这类中间件并不能支持事务机制，最终又会造成`DB`、缓存之间数据不一致。

2.3、先删缓存再写库



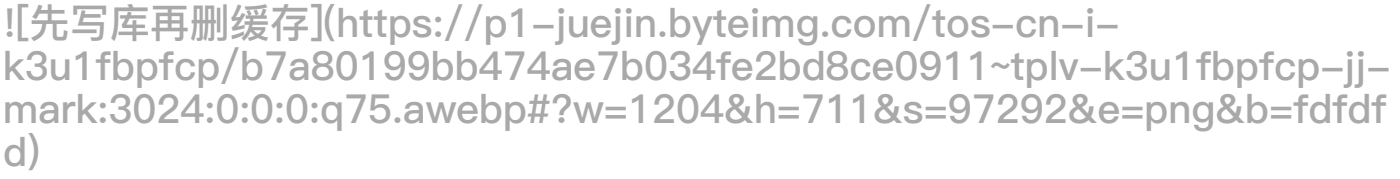
经过前面分析，只要处理写操作时，无论在写库前更新，还是后更新，总会因为各种原因导致不一致问题出现，那有没有好办法避免呢？有，**不更新缓存就好了，只需要在处理写操作时，将对应的缓存删除即可**。毕竟读取对应数据时，如果缓存里找不到，自动会去数据库里找，而后再将最新数据载入缓存。

好，既然要删缓存，那么我们来看看，先删缓存再写库有问题没？正常来说肯定没有，但从不正常的角度出发，那绝对有！上例子：

* ①请求`A`先将`Redis`里`userId=1（password=6666）`的缓存删除；
* ②请求`B`来读取`userId=1`的数据，没命中缓存，于是去库里查到老数据并塞进`Redis`；
* ③请求`A`将数据库里`userId=1`的数据，`password`值更改为`8888`。

仔细看，请求`A`刚删掉缓存里的数据，请求`B`就来读这条数据，这时没命中缓存就会查数据库，从库里又将`password=6666`的数据拿出来，并将其塞进`Redis`后返回；而请求`A`继续执行，会更新`password`字段为`8888`，数据库与缓存的数据又不一致了……

2.4、先写库再删缓存



既然先删缓存不行，那么先更新库再删缓存呢？有人看过这个说法：

> 如果先写库再删缓存，假设一条线程刚写完库，然后因为某些意外导致它挂了，因此没有执行删除缓存的动作，最终数据库、缓存的数据又不一致了……

上述这种情况，并不能`100%`成立，因为写操作一般都会开启事务，而删除缓存的代码通常和更新库的代码，都会写在一起（位于同一个方法）。如果一条线程刚写完库，然后就挂了，那么对应的事务肯定没有提交，这时数据库里变更的数据无法生效，对其他请求而言，数据库、缓存里看到的都是老值。

那么先写库再删缓存绝对安全吗？答案还是`No`，再来看例子：

* ①`Redis`中的缓存刚好过期；
* ②请求`A`正在将数据库的`password`值改为`8888`（还未提交事务）；
* ③请求`B`来获取`userId=1`的数据，未命中缓存，因此从库里拿到了旧值（还未放入`Redis`）；
* ④请求`A`删除缓存（执行完这个动作，请求`A`执行结束，事务提交）；
* ⑤请求`B`将拿到的旧值塞进`Redis`（这时数据库里已经是提交后的新值了）。

注意看这种特殊情况，一旦出现又会导致`MySQL、Redis`的数据不一致。不过这种情况几率很小，因为读请求通常比写请求要处理的快，很难复现“`B`先读旧值、`A`再删缓存、`B`再将旧值塞进缓存”这种现象。

除开上面这种特殊情况外，**先更新库再删缓存，如果删除缓存失败，也有可能造成`MySQL、Redis`的数据不一致**。因为库里的数据已经更新了，可是删除缓存时出现了意外，最终就导致缓存里的还是旧值。不过这种情况也很少发生，大家想想什么时候缓存会删除失败？`Redis`连接超时、`Redis`短暂故障、网络异常等情况，可这些情况一般都会抛出异常，而异常会导致事务回滚，`MySQL、Redis`的值又会是一样的了。

综上，只有在更新完数据库后，删除缓存时失败但没报错，才会导致数据产生不一致。又或者处理对应写操作时，没加事务控制（如分布式事务场景），删除缓存报错也不会使事务回滚，这也会造成数据不一致。

三、缓存一致性解决方案

经过前面的分析后，诸位会发现，不管是先操作缓存还是先操作库，总会存在不一致风险，究其根本原因，**就是因为操作数据库、操作缓存并不能保证原子性**，在没有涉及到跨库更新的场景中，可以将更新`DB`、删除缓存的逻辑放在一起，然后通过本地事务来保证两个动作的原子性。可是在分布式系统里，涉及到跨库更新、并且某个缓存聚合自多个服务时，就只能通过`XA-2PC、3PC`这种强一致的分布式事务来控制。

> “先写库再删缓存”方案结合事务控制，能彻底保证缓存和数据库的一致性，但会极大程度损耗性能。而且对于业务操作来说，执行业务逻辑、更新库都没报错，偏偏走到最后删缓存时出错，因此需要将整个事务回滚，这是极不公平的。

OK，既然通过事务保证原子性，才是一致性的最优解，那为什么还会有那么多花里胡哨的方案呢？如删除失败重试、延时双删、`Canel+MQ`异步更新……，下面一起来看看（这些方案都是建立在没有通过事务控制原子性的前提下）。

3.1、失败重试策略

这相当于“先写库再删缓存”的补充版，因为写完数据库之后，缓存不一定删成功，才会造成数据的不一致，所以只要保证删除缓存`100%`成功即可，怎么实现呢？伪代码如下：

```
public void updateInfo(DTO dto) {
    // 先更新数据库
    xxxDao.updateUserInfo(dto);

    try {
        // 删除缓存操作
        delCache(……);
    } catch(Exception e) {
        // 删除失败再次删除
        delCache(……);
    }
}
```

上面通过一个`try-catch`包住了删除缓存的操作，如果删除缓存时出现异常，则会再次重试删除。通过这种机制，能避免由于各种抖动因素导致的删除失败，毕竟很多时候第一次失败，再试一次时就会成功。

可这种方式，也不能`100%`保证一致性，因为第二次删除依旧可能失败，缓存里就还是旧数据。同时，因为删除失败重试机制，还是当前线程在执行，所以会使得请求的响应变慢。更好一点的做法是，将重试动作交给线程池，或者`MQ`来处理，不过这会让系统的复杂度变高。

3.2、延迟双删策略

延迟双删策略，从名字就能看出来，它会删两次，并且还会有延迟，伪逻辑如下：

```
public void updateInfo(DTO dto) {
    delCache(……);
    xxxDao.updateUserInfo(dto);
    Thread.sleep(500);
    delCache(……);
}
```

延迟双删的具体步骤就是先删一次缓存、再更新数据库、休眠一定时间、再删一次缓存，为啥要这么做呢？道理很简单，还记得前面“先写库再删缓存”的不一致问题吗？先删一次可以避免其他读请求读到旧数据，而更新完库再延迟一会儿删除，可以避免写库时出现的并发读，再次将旧数据载入进缓存。

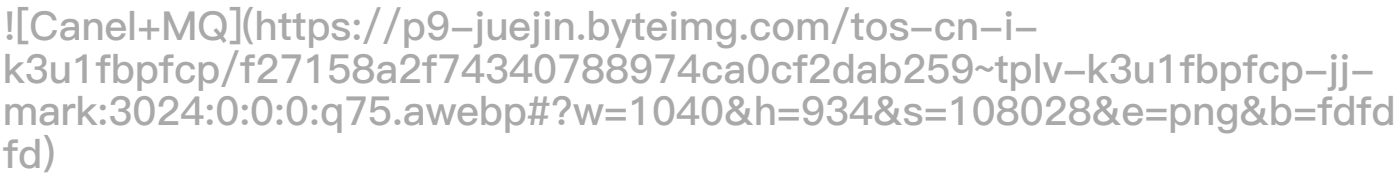
这种机制乍一听挺好，可仔细一推敲，就会发现有点华而不实，首先来看第一个问题，写库完成后需要延迟一定时间，请问哪个业务场景能接受这种延迟呢？除开自己写的`Demo`项目外，没有任何商业级项目能接受。当然，这里也可以用多线程优化，比如创建一个延迟线程池，把延迟删除的工作交给线程池即可。

> 引入线程池能避免请求阻塞，但会有新问题，比如线程池负荷较大时，删除缓存的延迟会比预设的更大等。

其次，第二次延迟删除如何保证`100%`成功？这依旧需要加入重试机制，否则同样会存在一致性问题。

3.3、异步更新/删除策略

不管是前面提到的删除重试机制，还是延迟双删策略，都存在一个致命问题：**对业务代码的侵入性太高了**，如何降低，或者做到代码零侵入呢？有人提到了一种机制，如下：



这种机制旧是通过`Canel`订阅`MySQL`的`bin-log`，因为`bin-log`日志包含了所有数据库变更操作，这时旧可以将增量更新推送给`MQ`，再由`MQ`来对变更过的数据进行“善后处理”，这个后置动作可以根据需求来自定义，如：

* ①再次对比缓存与库中的数据一致性，不一致则将缓存的数据删除；
* ②通过`MQ`的消费逻辑，将库中增量数据更新到缓存里；
* ……

这种方式最大的好处是对业务代码侵入性低，同时，只要保证`MQ`消费成功，对应的缓存就一定会被更新/删除。可是局限性也不算小，如果`MQ`的消费逻辑是更新缓存，那么对于简单的缓存数据处理起来并不费劲，但对于较为复杂的缓存数据，如数据看板的聚合信息、又或者落地页的组合信息等，这时`MQ`对应的消费逻辑就会很复杂……

还有，这种方式同样存在不一致风险，比如数据库更新后，因为需要订阅`Bin-log`日志来实现增量消费，尽管通过`Canel`来拉取日志，可依旧会存在数据延迟，而且`MQ`消费也需要时间。因此，在数据库更新后、`MQ`消费完成前，这属于不一致窗口（同时要保证`MQ`消费`100%`成功）。

3.4、更新标识策略

前面提到的各种方式，总会存在一些问题，而更新标识则是一种另辟蹊径的方法，逻辑如下：

```
public void updateInfo(DTO dto) {
    // 先设置更新标识
    redis.setFlag(……);

    // 对数据库进行更新
    xxxDao.updateUserInfo(dto);

    // 最后删除缓存
```

```
    delCache(key);
}

```

这种方案，其实就是在更新数据时，将对应数据的缓存，其`Value`置为一个特殊的更新标识，而读取数据的方法则需要做一定改动：

```
public XXX getInfo(Long userId) {
    Result result = redis.get(userId);

    if (Objects.nonNull(result)) {
        // 判断更新标识
        if (result.getFlag()) {
            // 如果在更新，就从库里查询数据并返回
            return db.getInfo(userId);
        }

        // 如果没在更新就直接返回缓存数据
        return result.getData();
    }

    // 如果未命中缓存，则获取库内数据并塞进缓存后返回
    XXX data = db.getInfo(userId);
    setCache(userId, data);
    return data;
}

```

上述伪代码的逻辑如下：

* ①先尝试从缓存里读取数据，然后判断是否命中缓存数据；
* ②如果命中了缓存，判断数据是否存在更新标识；
+ 如果数据正在更新，则直接从数据库获取数据并返回（不放入缓存）；
+ 如果数据未在更新，则直接返回获取到的缓存数据；
* ③如果未命中缓存，则从库里查询数据，并塞进缓存然后返回。

这种方案是在旁路缓存的基础之上，加上了一个“更新标识”的概念，这样就能降低数据更新时，由于并发读请求造成数据不一致的概率。同理，这种方式对代码的侵入性较高，需要对原有读写逻辑都进行改造，但相较于上一种方案，这种方案的成本会低很多~

四、缓存一致性问题总结

截止目前来说，我们对日常使用缓存的策略进行了展开叙述，可说来说去会发现，如果用了缓存，除非通过事务来控制原子性，否则就很难保证数据库与缓存的`100%`一致性，而提出的诸多方案，都是为了降低出现不一致的概率，并不能彻底预防数据不一致场景发生。

在分布式系统中，使用缓存是为了提升性能、承载并发，如果使用分布式事务、分布式锁来保证`MySQL、Redis`的双写原子性，就会极大程序上降低写操作的性能，这反而得不偿失。因此，如果想要保证绝对的一致性，那就不要对这种“一致性敏感”的数据进行缓存，例如账户余额、商品库存等等。

最后，对于文中提到的诸多策略与方案，大家稍微知道就行，重点记住里面的旁路缓存策略，以及缓存预热思想，其他的策略在实际工作中意义并不大，毕竟用上缓存的数据，本身就是读多写少、并且对一致性没那么敏感的信息，只要不一致状况不经常出现，那么就无伤大雅。

> 所有文章已开始陆续同步至公众号：**竹子爱熊猫**，想在上便捷阅读的小伙伴可搜索~

原文链接: https://juejin.cn/post/7373136303179792395