

21 | 赫赫有名的双刃剑：缓存（上）

2019-10-28 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 19:31 大小 13.41M



你好，我是四火。

从今天开始，我们将继续在基于 Web 的全栈技术上深挖，本章我们介绍持久层。缓存是全栈开发中非常重要的一环，因此我把它放到了数据持久化系列的开篇。

缓存使用好了，会是一把无比锋利的宝剑，特别对于性能的提升往往是立竿见影的；但使用不好就会严重影响系统运行，甚至因为数据一致性问题造成严重的数据错误。这一讲，我将为你介绍缓存的本质以及缓存的应用模式。

缓存的本质

工作中，我们可能每周都会谈到缓存，我们见过各种各样的缓存实现，网上也有各种各样的解释和定义。可是，你觉得，到底什么是缓存呢？

我认为，缓存，简单说就是为了节约对原始资源重复获取的开销，而将结果数据副本存放起来以供获取的方式。

首先，缓存往往针对的是“资源”。我们前面已经多次提到过，当某一个操作是“幂等”的和“安全”的（如果不记得的话请重看 [🔗\[第 04 讲\]](#)），那么这样的操作就可以被抽象为对“资源”的获取操作，那么它才可以考虑被缓存。有些操作不幂等、不安全，比如银行转账，改变了目标对象的状态，自然就难以被缓存。

其次，缓存数据必须是“重复”获取的。缓存能生效的本质是空间换时间。也就是说，将曾经出现过的数据以占据缓存空间的方式存放下来，在下一次的访问时直接返回，从而节约了通过原始流程访问数据的时间。有时候，某些资源的获取行为本身是幂等的和安全的，但实际应用上却不会“重复”获取，那么这样的资源是无法被设计成真正的缓存的。我们把一批数据获取中，通过缓存获得数据的次数，除以总的次数，得到的结果，叫做缓存的命中率。

再次，缓存是为了解决“开销”的问题。这个开销，可不只有时间的开销。虽然我们在很多情况下讲的开销，确实都是在时间维度上的，但它还可以是 CPU、网络、I/O 等一切资源。例如我们有时在 Web 服务中增加一层缓存，是为了避免了对原始资源获取的时候，对数据库资源调用的开销。

最后，缓存的存取其实不一定是“更快”的。有些程序员朋友对缓存访问总有一个比原始资源访问“更快”的概念，但这是不确切的。那不快，还要缓存干什么呢？别急，请往下看。

针对上面说的对“开销”的节约，你可以想象，每一种开销都能够成为缓存使用的动机。但其中，**有两个使用动机最为常见，一个是 latency，延迟**，即追求更低的延迟，这也是“更快”这个印象的由来；**另一个使用动机，是 throughput，吞吐量**，即追求更高的吞吐量。这个事实存在，也很常见，但是却较少人提及，且看下面的例子。

比如某个系统，数据在关系数据库中存放，获取速度很快，但是还在 S3 这个分布式文件系统上存放有数据副本，它的访问速度在该系统中要低于数据库的访问速度。某些请求量大的下游系统，会去 S3 获取数据，这样就缓和了前一条提到的数据库“开销”问题，但数据获取的速度却降下来了。这里 S3 存放的数据，也可以成为很有意义的缓存，即便它的存取其

实是更慢的。这种情况下，S3 并没有改善延迟，但提供了额外的吞吐量，符合上面提到的第二个使用动机。

另外，即便我们平时谈论的缓存“更快”访问的场景，**这个“快”也是相对而言的，在不同系统中同一对象会发生角色的变化。**例如，CPU 的多级高速缓存，就是内存访问的“缓存”；而内存虽然较 CPU 存取较慢，但比磁盘快得多，因此它可以被用作磁盘的“缓存”介质。

缓存无处不在

曾经有一个很经典的问题，讲的大致是当浏览器地址栏中，输入 URL（比如极客时间 <https://time.geekbang.org/>）按下回车，之后的几秒钟时间里，到底发生了什么。我们今天还来谈论这件事情，但是从一个特别的角度——缓存的角度来审视它。

对于地址栏中输入的域名，浏览器需要搞清楚它代表的 IP 地址，才能进行访问。过程如下：


它会先查询浏览器内部的“域名 -IP”缓存，如果你曾经使用该浏览器访问过这个域名，这里很可能留有曾经的映射缓存；

如果没有，会查询操作系统是否存在这个缓存，例如在 Mac 中，我们可以通过修改 `/etc/hosts` 文件来自定义这个域名到 IP 的映射缓存；

如果还没有，就会查询域名服务器（DNS，Domain Name System），得到对应的 IP 和可缓存时间。

Linux 或 Mac 系统中，你可以使用 `dig` 命令来查询：

```
1 dig time.geekbang.org
```

 复制代码

得到的信息中包含：

```
1 time.geekbang.org. 600 IN A 39.106.233.176
```

 复制代码

这是说这个 IP 地址就是极客时间对应的地址，可以被缓存 600 秒。

当请求抵达服务端，在 [反向代理](#) 中也是可以进行缓存配置的，比如我们曾经在 [\[第 09 讲\]](#) 中介绍过服务端包含 SSI 的方式来加载母页面上的一些静态内容。

接着，请求终于抵达服务端的代码逻辑了，对于一个采用 MVC 架构的应用来说，MVC 的各层都是可以应用缓存模式的。

对于 Controller 层来说，我们在 [\[第 12 讲\]](#) 中曾经介绍过拦截过滤器，而拦截过滤器中，我们就是可以配置缓存来过滤服务的，即满足某些要求的可缓存请求，我们可以直接通过过滤器返回缓存结果，而不执行后面的逻辑，我们在下一讲会学到具体怎样配置。

对于 Model 层来说，几乎所有的数据库 ORM 框架都提供了缓存能力，对于贫血模型的系统，在 DAO 上方的 Service 层基于其暴露的 API 应用缓存，也是一种非常常见的形式。

对于 View 层，很多页面模板都支持缓存标签，页面中的部分内容，不需要每次都执行渲染操作（这个开销很可能不止渲染本身，还包括需要调用模型层的接口而造成显著的系统开销），而可以直接从缓存中获取渲染后的数据并返回。

当母页面 HTML 返回了浏览器，还需要加载页面上需要的大量资源，包括 CSS、JavaScript、图像等等，都是可以通过读取浏览器内的缓存，而避免一个新的 HTTP 请求的开销的。通过服务端设置返回 HTTP 响应的 Cache-Control 头，就可以很容易做到这一点。例如：

```
1 Cache-Control: public, max-age=84600
```

 复制代码

上面这个请求头就是说，这个响应中的数据是“公有”的，可以被任意级节点（包括代理节点等等）缓存最多 84600 秒。

即便某资源无法被缓存，必须发起单独的 HTTP 请求去获取这样的资源，也可以通过 CDN 的方式，去较近的资源服务器获取，而这样的资源服务器，对于分布式网络远端的中心节点来说，就是它的缓存。

你看，对于这样的一个过程，居然有那么多缓存默默地工作，为你的网上冲浪保驾护航。如果继续往细了说，这个过程中你会看到更多的缓存技术应用，但我们就此打住吧，这些例子已经足够说明缓存应用的广泛度和重要性了。

缓存应用模式

在 Web 应用中，缓存的应用是有一些模式的，而我们可以归纳出这些模式以比较的方式来学习，了解其优劣，从而在实际业务中可以合理地使用它们。

1. Cache-Aside

这是最常见的一种缓存应用模式，整个过程也很好理解。

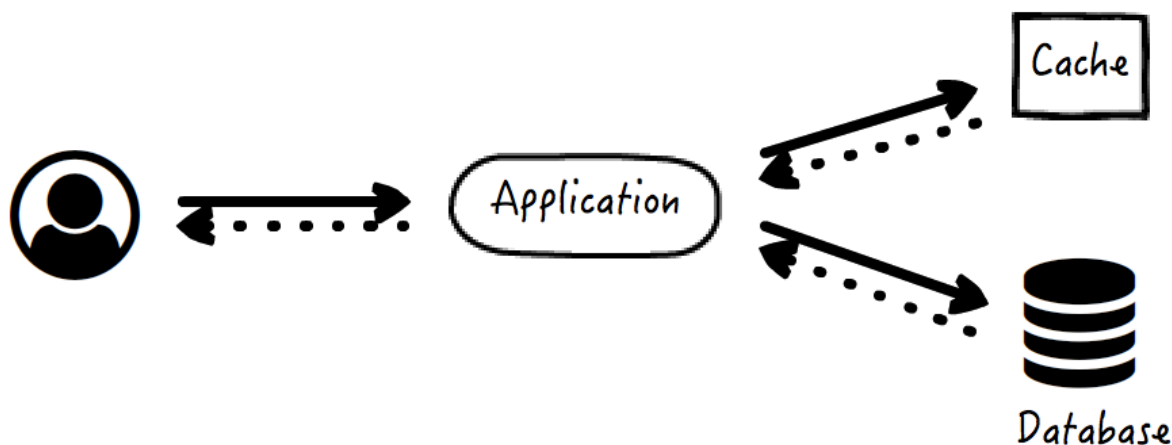
数据获取策略：

应用先去查看缓存是否有所需数据；

如果有，应用直接将缓存数据返回给请求方；

如果没有，应用执行原始逻辑，例如查询数据库得到结果数据；

应用将结果数据写入缓存。



我们见到的多数缓存，例如前面提到的拦截过滤器中的缓存，基本上都是按照这种方式来配置和使用的。

数据读取的异常情形：

如果数据库读取异常，直接返回失败，没有数据不一致的情况发生；

如果数据库读取成功，但是缓存写入失败，那么下一次同一数据的访问还将继续尝试写入，因此这时也没有不一致的情况发生。

可见，这两种异常情形都是“安全”的。

数据更新策略：

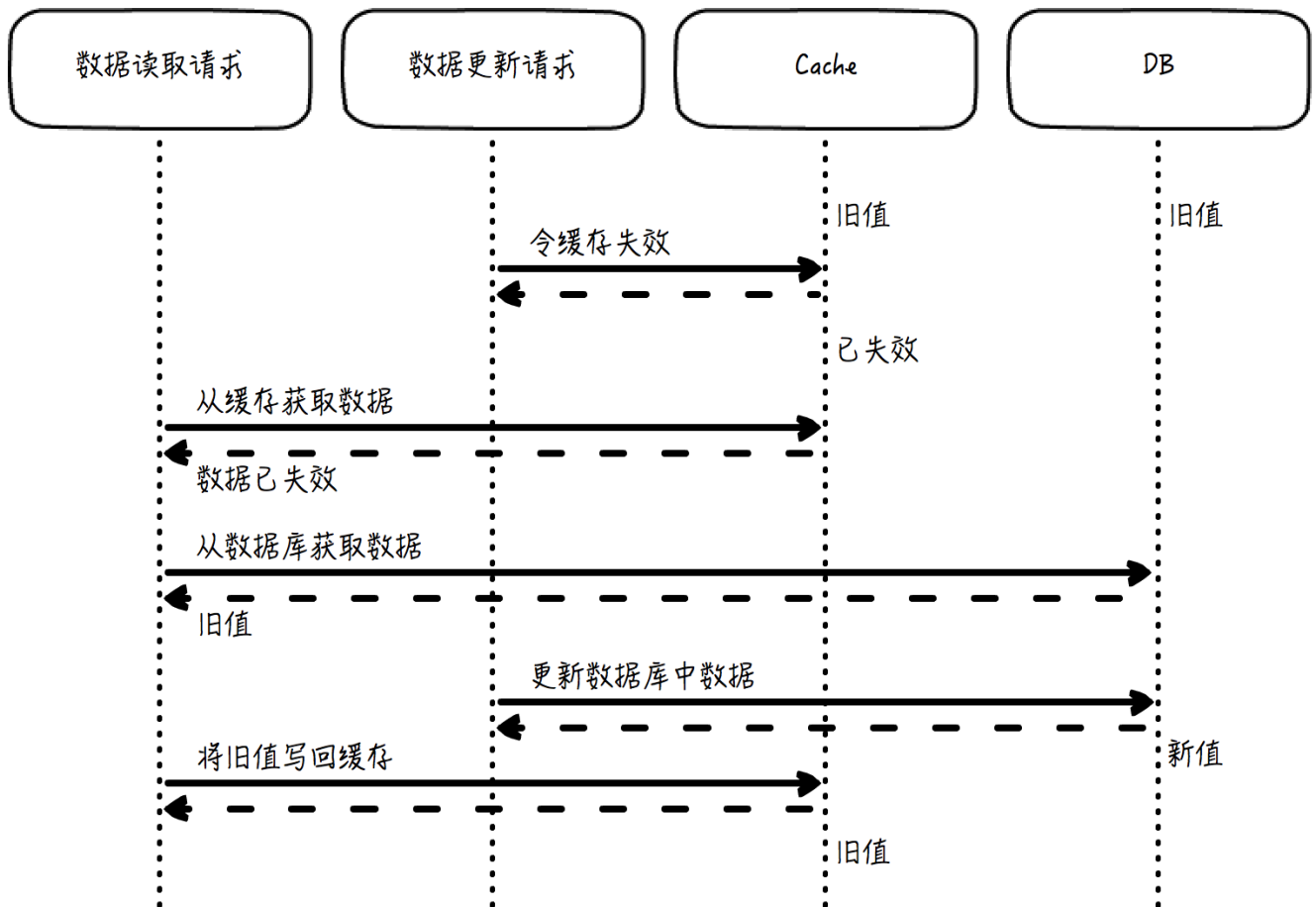
应用先更新数据库；

应用再令缓存失效。

这里，避免踩坑的关键点有两个：

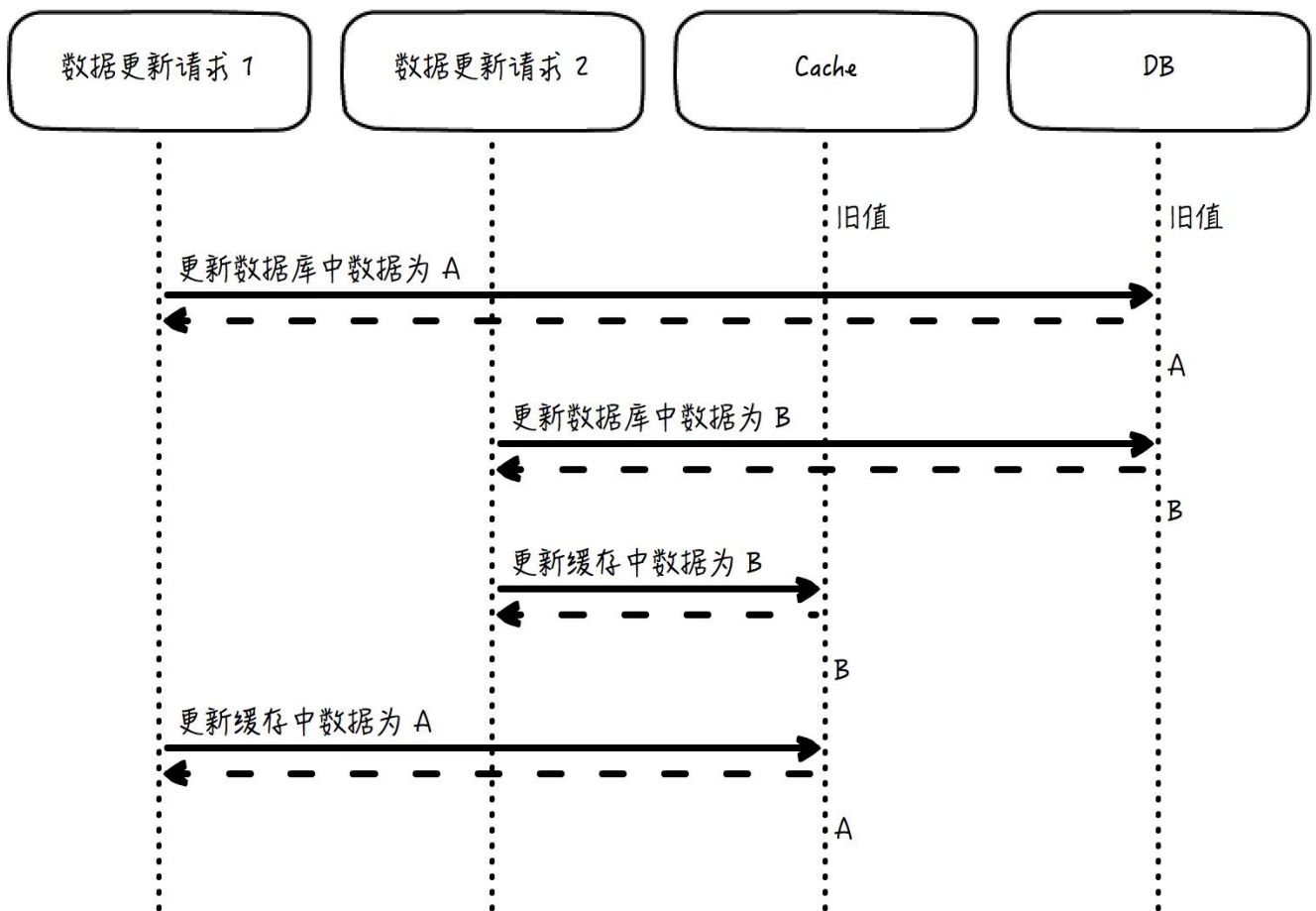
数据更新的这个策略，通常来说，最重要的一点是**必须先更新数据库，而不是先令缓存失效**，即这个顺序不能倒过来。原因在于，如果先令缓存失效，那么在数据库更新成功前，如果有另外一个请求访问了缓存，发现缓存数据库已经失效，于是就会按照数据获取策略，从数据库中使用这个已经陈旧的数值去更新缓存中的数据，这就导致这个过期的数据会长期存在于缓存中，最终导致数据不一致的严重问题。

这里我画了一张图，可以帮你理解，如果先令缓存失效，再更新数据库，为什么会导致问题：



第二个关键点是，**数据库更新以后，需要令缓存失效，而不是更新缓存为数据库的最新值。**为什么呢？你想一下，如果两个几乎同时发出的请求分别要更新数据库中的值为 A 和 B，如果结果是 B 的更新晚于 A，那么数据库中的最终值是 B。但是，如果在数据库更新后去更新缓存，而不是令缓存失效，那么缓存中的数据就有可能是 A，而不是 B。因为数据库虽然是“更新为 A”在“更新为 B”之前发生，但如果不做特殊的跨存储系统的事务控制，缓存的更新顺序就未必会遵从“A 先于 B”这个规则，这就会导致这个缓存中的数据会是一个长期错误的值 A。

这张图可以帮你理解，如果是更新缓存为数据库最新值，而不是令缓存失效，为什么会产生问题：



如果是令缓存失效，这个问题就消失了。因为 B 是后写入数据库的，那么在 B 写入数据库以后，无论是写入 B 的请求让缓存失效，还是并发的竞争情形下写入 A 的请求让缓存失效，缓存反正都是失效了。那么下一次的访问就会从数据库中取得最新的值，并写入缓存，这个值就一定是 B。

这两个关键点非常重要，而且不当使用引起的错误还非常常见，希望你可以完全理解它们。在我参与过的项目中，在这两个关键点上出错的系统我都见过（在这两点做到的情况下，其实还有一个理论上极小概率的情况下依然会出现数据错误，但是这个概率如此之小，以至于一般的系统设计当中都会直接将它忽略，但是你依然可以考虑一下它是什么）。

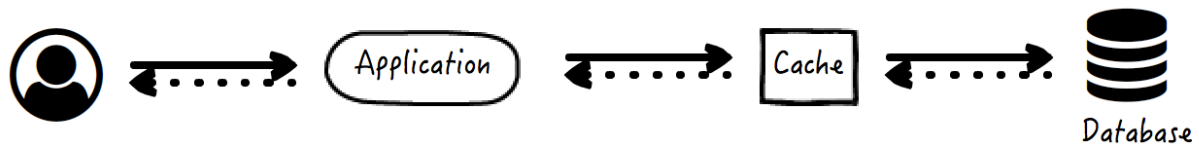
数据更新的异常情形：

如果数据库操作失败，那么直接返回失败，没有数据不一致的情况发生；

如果数据库操作成功，但是缓存失效操作失败，这个问题很难发生，但一旦发生就会非常麻烦，缓存中的数据是过期数据，需要特殊处理来纠正。

2. Read-Through

这种情况下缓存系统彻底变成了它身后数据库的代理，二者成为了一个整体，应用的请求访问只能看到缓存的返回数据，而数据库系统对它是透明的。



有的框架提供的内置缓存，例如一些 ORM 框架，就是按这种 Read-Through 和 Write-Through 来实现的。

数据获取策略：

应用向缓存要求数据；

如果缓存中有数据，返回给应用，应用再将数据返回；

如果没有，缓存查询数据库，并将结果写入自己；

缓存将数据返回给应用。

数据读取异常的情况分析和 Cache-Aside 类似，没有数据不一致的情况发生。

3. Write-Through

和 Read-Through 类似，图示同上，但 Write-Through 是用来处理数据更新的场景。

数据更新策略：

应用要求缓存更新数据；

如果缓存中有对应数据，先更新该数据；

缓存再更新数据库中的数据；

缓存告知应用更新完成。

这里的一个关键点是，**缓存系统需要自己内部保证并发场景下，缓存更新的顺序要和数据库更新的顺序一致**。比如说，两个请求分别要把数据更新为 A 和 B，那么如果 B 后写入数据库，缓存中最后的结果也必须是 B。这个一致性可以用乐观锁等方式来保证。

数据更新的异常情形：

如果缓存更新失败，直接返回失败，没有数据不一致的情况发生；

如果缓存更新成功，数据库更新失败，这种情况下需要回滚缓存中的更新，或者干脆从缓存中删除该数据。

还有一种和 Write-Through 非常类似的数据更新模式，叫做 Write-Around。它们的区别在于 Write-Through 需要更新缓存和数据库，而 Write-Around 只更新数据库（缓存的更新完全留给读操作）。

4. Write-Back

对于 Write-Back 模式来说，更新操作发生的时候，数据写入缓存之后就立即返回了，而数据库的更新异步完成。这种模式在一些分布式系统中很常见。

这种方式带来的最大好处是拥有最大的请求吞吐量，并且操作非常迅速，数据库的更新甚至可以批量进行，因而拥有杰出的更新效率以及稳定的速率，这个缓存就像是一个写入的缓冲，可以平滑访问尖峰。另外，对于存在数据库短时间无法访问的问题，它也能够很好地处理。

但是它的弊端也很明显，异步更新一定会存在着不可避免的一致性问题，并且也存在着数据丢失的风险（数据写入缓存但还未入库时，如果宕机了，那么这些数据就丢失了）。

总结思考

今天我们学习了缓存的本质、应用，仔细比较了几种常见的应用模式。在理解缓存本质的基础上，Cache-Aside 模式是缓存应用模式中的重点，在我们实际系统的设计和实现中，它是最为常用的那一个。希望这些缓存的知识可以帮到你！

现在我来提两个问题，检验一下今天的学习成果吧。

在你参与的项目中，是否应用到了缓存，属于哪一个应用模式，能否举例说明呢？

这一讲提到了几种缓存应用模式，你能否说出 Cache-Aside 和 Write-Back 这两种模式各有什么优劣，它们都适应怎样的实际场景呢？

看到最后，你可能会想，不是说双刃剑吗？杀敌的那一刀已经介绍了，可自伤的那一刀呢？别急，我们下一讲就会讲到缓存使用中的坑，以期有效避免缓存使用过程中的问题。今天的内容就到这里，欢迎你和我讨论。

扩展阅读

文中提到使用 dig 命令来查询 DNS 返回的 IP 地址，想了解更完整的原理，可以参阅 [DNS 原理入门](#)。

文中提到了 HTTP 响应中的缓存设置头，请参阅 MDN 的 [HTTP 缓存](#) 一节以获得更为细致的讲解。

文中提到了乐观锁，不清楚的话，你可以阅读这个 [词条](#)，以及这篇 [文章](#) 以进一步理解。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



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。

精选留言 (1)

写留言



leslie

2019-10-28

数据写入缓存故而这就是日志的问题啊：现在其实许多数据库模型同样宕机就OVER，纠其根本原因还是许多的不严谨；日志是灾难恢复的主要手段。

数据库内部查询同样是先看缓存是否有，没有再去重新查找；毕竟缓存速度>内存>磁盘。异步最大的难题是是事务性：这块非关系型数据库处理的都不好。

展开 ∨

2

