页面优化技术

1. **页面缓存+URL缓存+对象缓存**
   由于并发瓶颈在数据库，想办法如何减少对数据库的访问，所以加若干缓存来提高，通过各种粒度的缓存，最大粒度页面缓存到最小粒度的对象级缓存。

页面缓存

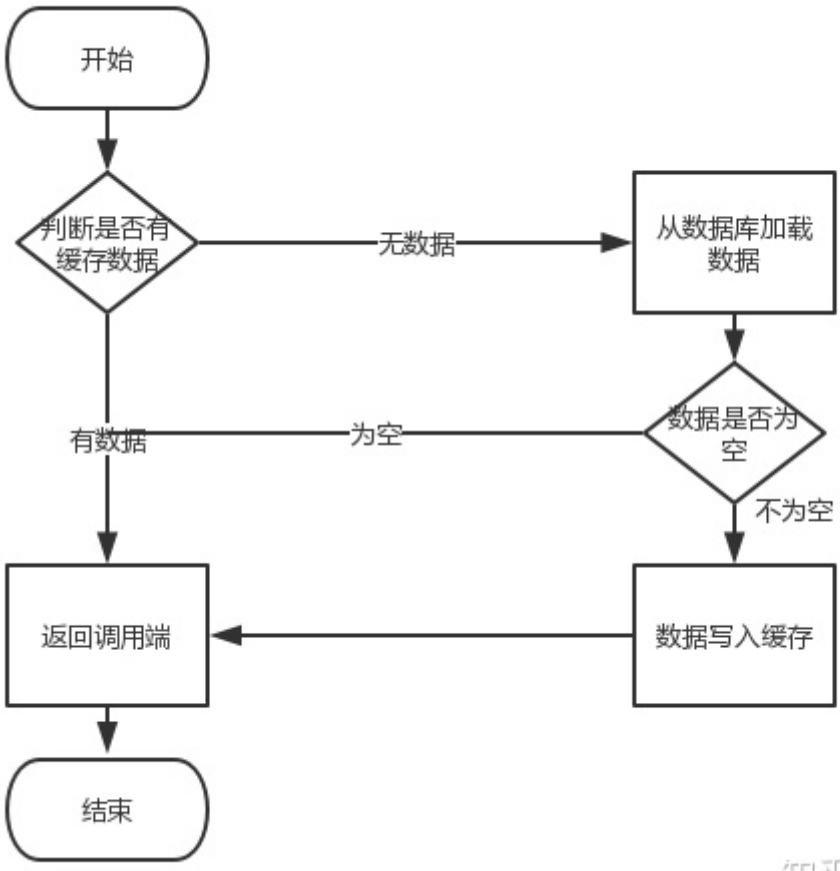
1.取缓存 （缓存里面存的是html）
2.手动渲染模板
3.结果输出（直接输出html代码）

当访问goods_list页面的时候，如果从缓存中取到就返回这个html,(这里方法的返回格式已经设置为text/html，这样就是返回html的源代码)，如果取不到，利用ThymeleafViewResolver的getTemplateEngine().process和我们获取到的数据，渲染模板，并且在返回到前端之前保存至缓存里面，然后之后再来获取的时候，只要缓存里面存的goods_list页面的html还没有过期，那么直接返回给前端即可。

一般这个页面缓存时间，也不会很长，防止数据的时效性很低。但是可以防止短时间大并发访问。

GoodsKey :作为页面缓存的缓存Key的前缀，缓存有效时间，一般设置为1分钟

对象缓存



相比页面缓存是更细粒度缓存。在实际项目中， 不会大规模使用页面缓存，对象缓存就是当用到用户数据的时候，可以从缓存中取出。比如：更新用户密码，根据token来获取用户缓存对象。

MiaoshaUserService里面增加getById方法，**先去取缓存，如果缓存中拿不到，那么就去取数据库，然后再设置到缓存中去**

更新用户密码：更新数据库与缓存，一定保证数据一致性，修改token关联的对象以及id关联的对象，先更新数据库后删除缓存，不能直接删除token，删除之后就不能登录了，再将token以及对应的用户信息一起再写回缓存里面去。

Redis与Mysql双写一致性方案解析(更新问题)

给缓存设置过期时间，是保证最终一致性的解决方案。这种方案下，我们可以对存入缓存的数据设置过期时间，所有的写操作以数据库为准，对缓存操作只是尽最大努力即可。也就是说如果数据库写成功，缓存更新失败，那么只要到达过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。因此，接下来讨论的思路不依赖于给缓存设置过期时间这个方案。 首先:

为什么不能先处理缓存，再更新数据库呢？
为什么你的缓存更新策略是先更新数据库后删除缓存，讲讲其他的情况有什么问题？

问题：怎么保持缓存与数据库一致？

要解答这个问题，我们首先来看不一致的几种情况。我将不一致分为三种情况

数据库有数据，缓存没有数据；
数据库有数据，缓存也有数据，数据不相等；
数据库没有数据，缓存有数据。

策略：

1. **首先尝试从缓存读取，读到数据则直接返回；如果读不到，就读数据库，并将数据会写到缓存，并返回。**
2. **需要更新数据时，先更新数据库，然后把缓存里对应的数据失效掉（删掉）。**

   也就是说
3. **失效**：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。
4. **命中**：应用程序从cache中取数据，取到后返回。
5. **更新**：先把数据存到数据库中，成功后，再让缓存失效。

1 先删除缓存，然后再更新数据库。这么做引发的问题是

- （1）请求A进行写操作，删除缓存
- （2）请求B查询发现缓存不存在
- （3）请求B去数据库查询得到旧值
- （4）请求B将旧值写入缓存
- （5）请求A将新值写入数据库 上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

解决方法:

- （1）先淘汰缓存
- （2）再写数据库（这两步和原来一样）
- （3）休眠1秒，再次淘汰缓存 这么做，可以将1秒内所造成的缓存脏数据，再次删除。

另外有人会问，如果采用你提到的方法，为什么最后是把缓存的数据删掉，而不是把更新的数据写到缓存里(

先更新数据库,再更新缓存)。这么做引发的问题是，

原因一（线程安全角度） 同时有请求A和请求B进行更新操作，那么会出现

- （1）线程A更新了数据库
- （2）线程B更新了数据库
- （3）线程B更新了缓存
- （4）线程A更新了缓存

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据，因此不考虑。

原因二（业务场景角度） 有如下两点：

- （1）如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。
- （2）如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

我想出的解决方案大概有以下几种：

1. **对删除缓存进行重试，数据的一致性要求越高，我越是重试得快。**
2. **定期全量更新，简单地说，就是我定期把后再全量加载。**
3. **给所有的缓存一个失效期。**

第三种方案可以说是一个大杀器，任何不一致，都可以靠失效期解决，失效期越短，数据一致性越高。但是失效期越短，查数据库就会越频繁。因此失效期应该根据业务来定。

分布式CAP原理

高可用的数据

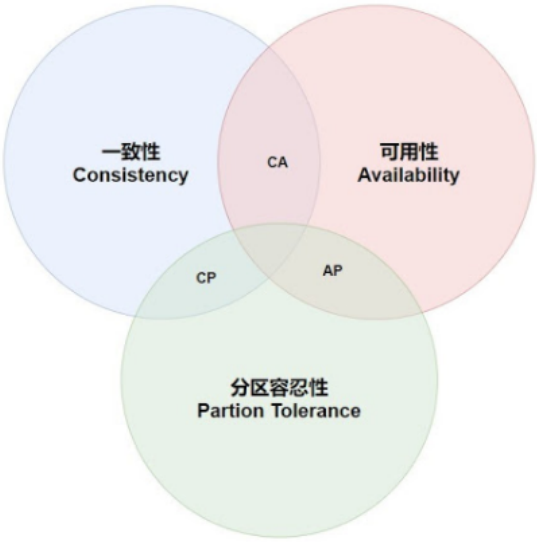
CAP理论是分布式系统中对数据的管理而形成一套理论知识，CAP是设计分布式系统所必须考虑的架构问题。对于CAP本身可以解释如下：

- Consistency(一致性)： 数据一致更新，所有数据变动都是同步的
- Availability(可用性)： 好的响应性能
- Partition tolerance(分区耐受性)： 可靠性

上面的解释可能显得太过抽象，举例来说在高可用的网站架构中，对于数据基础提出了以下的要求：

- **分区耐受性**
  保证数据可持久存储，在各种情况下都不会出现数据丢失的问题。为了实现数据的持久性，不但需要在写入的时候保证数据能够持久存储，还需要能够将数据备份一个或多个副本，存放在不同的物理设备上，防止某个存储设备发生故障时，数据不会丢失。
- **数据一致性**
  在数据有多份副本的情况下，如果网络、服务器、软件出现了故障，会导致部分副本写入失败。这就造成了多个副本之间的数据不一致，数据内容冲突。
- **数据可用性**
  多个副本分别存储于不同的物理设备的情况下，如果某个设备损坏，就需要从另一个数据存储设备上访问数据。如果这个过程不能很快完成，或者在完成的过程中需要停止终端用户访问数据，那么在切换存储设备的这段时间内，数据就是不可访问的。

CAP原理认为，一个提供数据服务的存储系统无法同时完美的满足一致性（Consistency）、数据可用性（Availability）、分区耐受性（Partition Tolerance）这三个条件。三者的关系：



在实际的大型网络应用中，数据的规模会快速扩张，因此数据架构的伸缩性（分区耐受性P）必不可少。当规模变大之后，机器的数量也会增大，这时网络和服务器故障会更频繁出现，想要保证应用可用，就必须保证分布式处理系统的高可用性A。**所以在大型网站中，通常会选择强化分布式存储系统的可用性（A）和伸缩性（P）**，在某种程度上放弃一致性（C）。**一般来说，数据不一致的情况通常出现在高并发写操作或者集群状态不稳（故障恢复，集群扩容...）的情况下，应用系统需要对分布式数据处理系统的数据不一致性有一定的了解并进行某种意义上的补偿工作，以避免应用出现数据不正确。**

CAP原理对于可伸缩的分布式系统设计具有重要的意义，在系统设计开发过程中，不恰当的迎合各种需求，企图打造一个完美的产品，可能会使设计进入两难之地，难以为继。
具体来说，数据一致性C又可分为以下几点：

- **数据强一致**
  各个副本中的数据总是强一致的。这种设计正确性很高，但是会在一定程度上损耗性能。
- **数据用户一致**
  应用访问数据时通过一定的纠错和校验机制，把多个数据可能不一致的副本的数据综合计算返回一个一致且确定的数据给用户。大型互联网架构一般采用这种设计，性能较好，并且数据不会出现错误。
- **数据最终一致**
  物理存储的数据不一致，用户访问得到的数据也可能不一致，但经过一段时间的自我修正（通常很短时间），数据会达到最终一致。该设计性能最高，但可能有数据错误。

因为很难去同时满足CAP，大型网站通常会综合成本、技术、业务场景等条件，结合应用服务和其它的数据监控与纠错功能，使存储系统达到用户一致，保证用户最终访问数据的正确性。