软件架构与中间件

涂志莹 tzy_hit@hit.edu.cn

苏统华 thsu@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第4章

数据层的软件架构技术

第4章 数据层的软件架构技术

4.1 数据驱动的软件架构演化

4.2 数据读写与主从分离

4.3 数据分库分表

4.4 数据缓存

4.5 非关系型数据库

4.6 数据层架构案例

4.4
数据缓存

思考：数据缓存在哪里？

4.4
数据缓存

1、数据缓存的基本理论

2、本地缓存

3、分布式缓存

4、缓存问题讨论

4.4.1 数据缓存的基本理论

- 数据缓存的基本概念
- 数据缓存的基本操作

数据缓存的基本概念

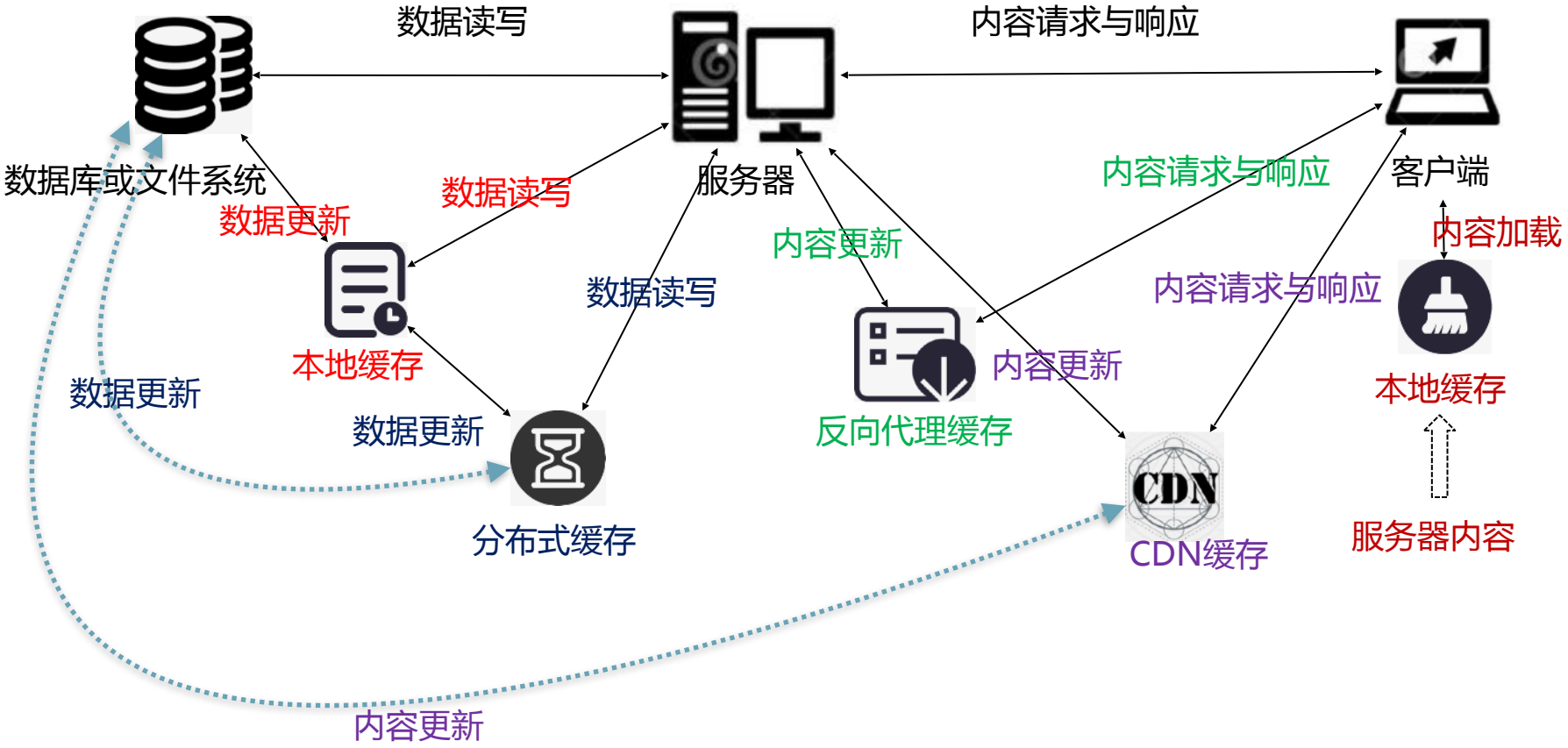
数据缓存的基本定义

- 缓存的概念：
  - 用于存储数据的硬件或软件组件，以使得后续更快访问响应的数据。
  - 缓存中的数据可能是提前计算好的结果、数据的副本等。
- 缓存的作用：
  - 主要解决高并发，热点数据访问的性能问题。
  - 提供高性能的数据快速访问。
- 缓存的原理：
  - 将数据写入到读取速度更快的存储
  - 将数据缓存到离应用最近的位置
  - 将数据缓存到离用户最近的位置

数据缓存的存储介质

- 虽然从硬件介质上来看，无非就是内存和硬盘两种，但从技术上，可以分成内存、硬盘文件、数据库。
  - 内存：缓存于内存中是最快的选择，无需额外的I/O开销，但是内存的缺点是没有持久化到物理磁盘，一旦应用异常break down而重新启动，数据很难或者无法复原。
  - 硬盘：一般来说，很多缓存框架会结合使用内存和硬盘，在内存分配空间满了或是在异常的情况下，可以被动或主动的将内存空间数据持久化到硬盘中，达到释放空间或备份数据的目的。
  - 数据库：非传统数据库，而是key-value存储结构的特殊数据库（如BerkeleyDB和Redis），响应速度和吞吐量都远高于关系型数据库等。

数据缓存的基本架构


数据读写
内容请求与响应
数据库或文件系统
服务器
客户端
数据更新
数据读写
内容更新
内容请求与响应
内容加载
数据读写
内容请求与响应
本地缓存
内容更新
数据更新
本地缓存
数据更新
反向代理缓存
CDN
分布式缓存
CDN缓存
服务器内容
内容更新

数据缓存的基本类型

- 1、本地缓存
- 2、分布式缓存
- 3、反向代理缓存
- 4、CDN缓存

数据缓存的基本类型-本地缓存

- 本地缓存指的是在应用中的缓存组件，其最大的优点是应用和cache是在同一个进程内部，请求缓存非常快速，没有过多的网络开销等，在单应用不需要集群支持或者集群情况下各节点无需互相通知的场景下使用本地缓存较合适；
- 本地缓存的缺点：因为缓存跟应用程序耦合，多个应用程序无法直接的共享缓存，各应用或集群的各节点都需要维护自己的单独缓存，对内存是一种浪费。
- 应用场景：缓存字典等常用数据。
- 缓存介质：
  - 硬盘缓存：将数据缓存到硬盘，减少网络传输的消耗
  - 内存缓存：直接将数据存储到本级内存中，通过程序直接维护缓存对象。
- 实现方法：应用编码；中间件，如Ehcache、 Guava Cache等

数据缓存的基本类型-分布式缓存

- 分布式缓存指的是与应用分离的缓存组件或服务，其最大的优点是自身就是一个独立的应用，与本地应用隔离，多个应用可直接的共享缓存。
- 应用场景：
  - 缓存经过复杂运算得出的数据
  - 缓存存储系统中频繁访问的热点数据，减轻存储系统压力
- 常用的分布式缓存中间件：
  - Memcached
  - Redis

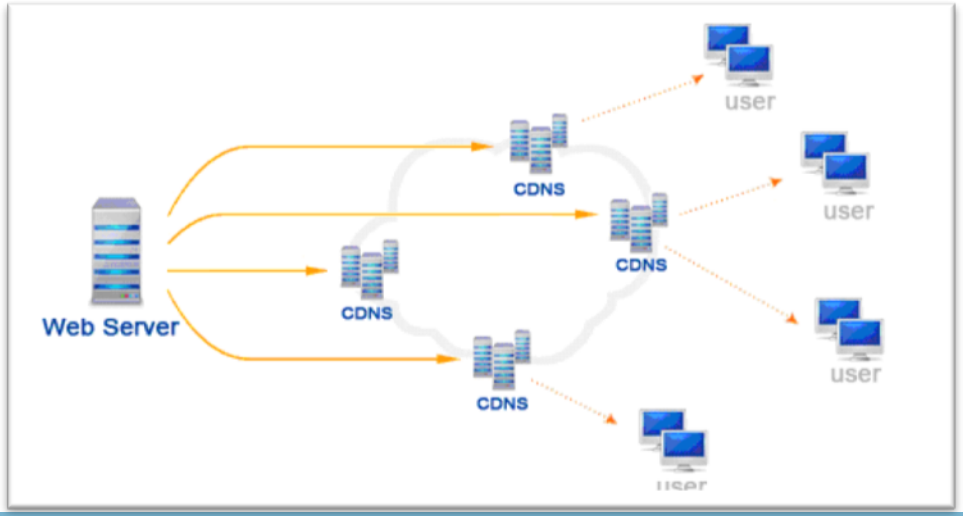
数据缓存的基本类型-反向代理缓存

- 反向代理位于应用服务器机房，处理所有对WEB服务器的请求。
  如果用户请求的页面在代理服务器上有缓冲的话，代理服务器直接将缓冲内容发送给用户。如果没有缓冲则先向WEB服务器发出请求，取回数据，本地缓存后再发送给用户。通过降低向WEB服务器的请求数，从而降低了WEB服务器的负载。
- 应用场景：
  - 一般只缓存体积较小的静态文件资源，如css、js、图片
- 常用的开源实现：
  - Varnish
  - Nginx
  - Squid

数据缓存的基本类型-CDN缓存

- CDN (Content Delivery Network)：内容分发网络
  - 通过在现有互联网中增加一层新的网络架构（CDNS），将网站内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容。
  - CDN目标：解决由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题。

- CDN：将一个服务器的内容平均分布到多个服务器上；智能识别服务器，让用户获取离用户最近的服务器，提高访问速度。
  - 分布式存储
  - 负载均衡
  - 网络请求的重定向
  - 内容管理



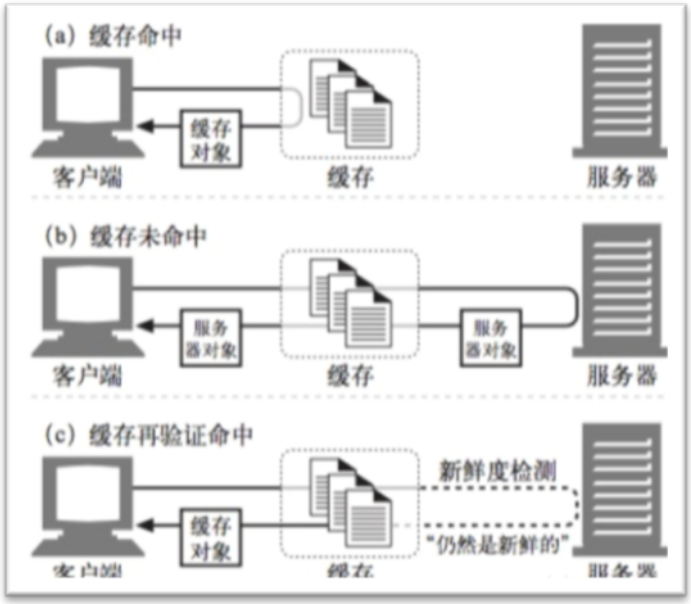
数据缓存的术语定义

- 命中率
  - 命中率=返回正确结果数/请求缓存次数
  - 命中率问题是缓存中的一个非常重要的问题，它是衡量缓存有效性的重要指标。命中率越高，表明缓存的使用率越高。
  - 缓存的管理者希望缓存命中率接近 100%。而实际得到的命中率则与缓存的大小、缓存用户兴趣点的相似性、缓存数据的变化或个性化频率，以及如何配置缓存有关。
  - 命中率很难预测，但对现在中等规模的Web缓存来说，40%的命中率是很合理的。

- 最大元素（或最大空间）
  - 缓存中可以存放的最大元素的数量，一旦缓存中元素数量超过这个值（或者缓存数据所占空间超过其最大支持空间），那么将会触发缓存启动清空策略
  - 根据不同的场景合理的设置最大元素值往往可以一定程度上提高缓存的命中率，从而更有效的使用缓存。

数据缓存的基本操作

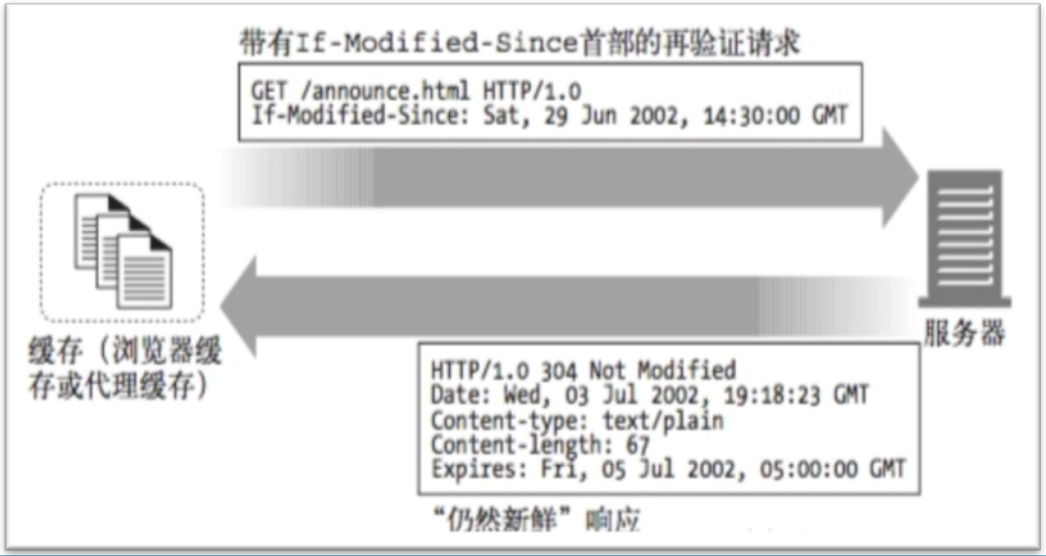
数据缓存命中与验证



- HTTP 再验证 (revalidation)：原始服务器的内容可能会发生变化，缓存要不时对其进行检测，看看它们保存的副本是否仍是服务器上最新的副本，进行“新鲜度检测”。
- 缓存可以在任意时刻，以任意的频率对副本进行再验证。但由于缓存中通常会包含数百万的文档，而且网络带宽是很珍贵的，所以大部分缓存只有在客户端发起请求，并且副本旧得足以需要检测的时候，才会对副本进行再验证。
- 为了有效地进行再验证，HTTP 定义了一些特殊的请求，不用从服务器上获取整个对象，就可以快速检测出内容是否是最新的。
- HTTP 为我们提供了几个用来对已缓存对象进行再验证的工具，但最常用的是 If-Modified-Since 首部。将这个首部添加到 GET 请求中去，就可以告诉服务器，只有在缓存了对象的副本之后，又对其进行了修改的情况下，才发送此对象。

数据缓存命中与验证

- 服务器收到 GET If-Modified-Since 请求时的 3 种情况:
  - 再验证命中(revalidate hit)或缓慢命中(slow hit)：如果服务器对象未被修改，服务器会向客户端发送一个小的 HTTP 304 Not Modified 响应。只要缓存知道副本仍然有效，就会再次将副本标识为暂时新鲜的，并将副本提供给客户端。
  - 再验证未命中：如果服务器对象与已缓存副本不同，服务器向客户端发送一条普通的、带有完整内容的 HTTP 200 OK 响应。
  - 对象被删除：如果服务器对象已经被删除了，服务器就回送一个 404 Not Found 响应，缓存也会将其副本删除。



数据缓存的基本清洗策略

- 缓存清空策略：在缓存的存储空间有限制，当缓存空间被用满时，既要保证稳定服务，又要有效提升命中率。常见的一般策略有：
  - FIFO(first in first out)：先进先出策略，最先进入缓存的数据在缓存空间不够的情况下（超出最大元素限制）会被优先被清除掉，以腾出新的空间接受新的数据。策略算法主要比较缓存元素的创建时间。在数据实效性要求场景下可选择该类策略，优先保障最新数据可用。
  - LFU(less frequently used)：最少使用策略，无论是否过期，根据元素的被使用次数判断，清除使用次数较少的元素释放空间。策略算法主要比较元素的hitCount（命中次数）。在保证高频数据有效性场景下，可选择这类策略。
  - LRU(least recently used)：最近最少使用策略，无论是否过期，根据元素最后一次被使用的时间戳，清除最远使用时间戳的元素释放空间。策略算法主要比较元素最近一次被get使用时间。在热点数据场景下较适用，优先保证热点数据的有效性。

设计适合自身数据特征的清空策略能有效提升命中率。

数据缓存的其他清洗策略

- 根据过期时间判断，清理过期时间最长的元素
- 根据过期时间判断，清理最近要过期的元素
- 随机清理
- 根据关键字（或元素内容）长短清理等

数据缓存的更新策略

- Cache aside
  - 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。
  - 命中：应用程序从cache中取数据，取到后返回。
  - 更新：先把数据存到数据库中，成功后，再让缓存失效。
- Read/Write Through Pattern
  - Read Through 是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或LRU换出），Cache Aside是由调用方负责把数据加载入缓存，而Read Through则用缓存服务自己来加载，从而对应用方是透明的。
  - Write Through 在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由Cache自己更新数据库（这是一个同步操作）。

数据缓存的更新策略

- Write Behind Caching Pattern
  - 俗称write back，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的I/O操作飞快无比，因为异步（比如消息队列），write back还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。
  - 但是这个设计的最大致命问题在于数据的非强一致性，极可能造成数据的丢失。假如使用redis作为缓存数据库，最致命的问题在于redis并不能保证绝对不丢失数据，也就是redis的持久化能力（两种持久化都无法保证数据绝对丢失）不足，redis一旦挂了，可能造成数据丢失且无法恢复。

4.4
数据缓存

1、数据缓存的基本概念

2、本地缓存

3、分布式缓存

4、缓存问题讨论

4.4.2 本地缓存

- 基础知识回顾
- 本地缓存的实现方法

基础知识回顾

JVM的内存管理

- JVM可以使用的内存分外2种：堆内内存（on-heap）和堆外内存（off-heap）
  - 堆内内存：完全由JVM负责分配和释放，如果程序导致内存泄露，那么就会遇到java.lang.OutOfMemoryError错误。
  - 堆外内存：JDK5.0之后为了能直接分配和释放内存，提高效率，允许代码直接操作本地内存。方式有2种：使用未公开的Unsafe和NIO包下ByteBuffer。
  - 使用ByteBuffer分配本地内存非常简单，直接ByteBuffer.allocateDirect(10 * 1024 * 1024)即可。

```
import java.nio.ByteBuffer;

public class TestDirectByteBuffer
{
    // -verbose:gc -XX:+PrintGCDetails -XX:MaxDirectMemorySize=40M
    public static void main(String[] args) throws Exception
    {
        while (true)
        {
            ByteBuffer buffer = ByteBuffer.allocateDirect(10 * 1024 * 1024);
        }
    }
}
```

堆内内存

- 堆Heap是内存中动态分配对象存在的地方。如果使用new一个对象，它就被分配在堆内存上。
- 一般情况下，Java中分配的非空对象都是由JVM的垃圾收集器管理的，也称为堆内内存。虚拟机会定期对垃圾内存进行回收，在某些特定的时间点，它会进行一次彻底的回收。
- 彻底回收时，垃圾收集器会对所有分配的堆内内存进行完整的扫描，这意味着一次垃圾收集对Java应用造成的影响，跟堆的大小是成正比的。过大的堆会影响Java应用的性能。

堆外内存

- 堆外内存意味着把内存对象分配在JVM的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。使用堆外内存能够降低JVM垃圾回收导致的暂停。
- 堆外内存有以下特点：
  - 对于大内存有良好的伸缩性
  - 对垃圾回收停顿的改善可以明显感觉到
  - 在进程间可以共享，减少虚拟机间的复制
- 堆外内存的问题有：
  - 数据结构不直观
  - 数据结构比较复杂，需要串行化（serialization），而串行化本身也会影响性能
  - 使用更大的内存，需要考虑虚拟内存（即硬盘）的速度带来的影响
  - 当对象从堆中脱离出来序列化，然后存储在一大块内存中，这就像它存储到磁盘上上一样，但它仍然在RAM中。对象在这种状态下不能直接使用，它们必须首先反序列化。虽不受垃圾收集影响，但序列化和反序列化会影响性能。

本地缓存的实现方法

编程直接实现缓存

- 成员变量或局部变量实现

```
1   public void UseLocalCache(){
2       //一个本地的缓存变量
3       Map<String, Object> localCacheStoreMap = new HashMap<String, Object>();
4
5       List<Object> infosList = this.getInfoList();
6       for(Object item:infosList){
7           if(localCacheStoreMap.containsKey(item)){ //缓存命中 使用缓存数据
8               // todo
9           } else { // 缓存未命中  IO获取数据，结果存入缓存
10              Object valueObject = this.getInfoFromDB();
11              localCacheStoreMap.put(valueObject.toString(), valueObject);
12
13          }
14      }
15  }
16  //示例
17  private List<Object> getInfoList(){
18      return new ArrayList<Object>();
19  }
20  //示例数据库IO获取
21  private Object getInfoFromDB(){
22      return new Object();
23  }
```

编程直接实现缓存

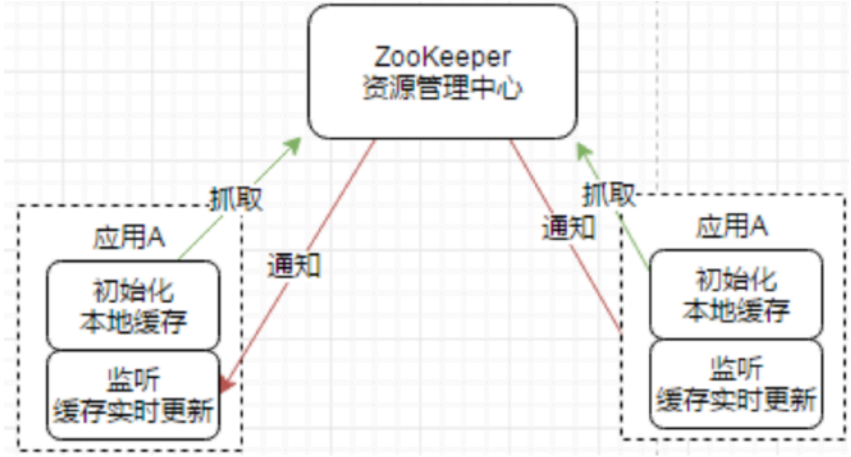
- 静态变量实现

```
1   public class CityUtils {
2       private static final HttpClient httpClient = ServerHolder.createClientWithPool();
3       private static Map<Integer, String> cityIdNameMap = new HashMap<Integer, String>();
4       private static Map<Integer, String> districtIdNameMap = new HashMap<Integer, String>();
5
6       static {
7           HttpGet get = new HttpGet("http://gis-in.sankuai.com/api/location/city/all");
8           BaseAuthorizationUtils.generateAuthAndDateHeader(get,
9                   BaseAuthorizationUtils.CLIENT_TO_REQUEST_MDC,
10                  BaseAuthorizationUtils.SECRET_TO_REQUEST_MDC);
11          try {
12              String resultStr = httpClient.execute(get, new BasicResponseHandler());
13              JSONObject resultJo = new JSONObject(resultStr);
14              JSONArray dataJa = resultJo.getJSONArray("data");
15              for (int i = 0; i < dataJa.length(); i++) {
16                  JSONObject itemJo = dataJa.getJSONObject(i);
17                  cityIdNameMap.put(itemJo.getInt("id"), itemJo.getString("name"));
18              }
19          } catch (Exception e) {
20              throw new RuntimeException("Init City List Error!", e);
21          }
22      }
23      static {
24          HttpGet get = new HttpGet("http://gis-in.sankuai.com/api/location/district/all");
25          BaseAuthorizationUtils.generateAuthAndDateHeader(get,
26              BaseAuthorizationUtils.CLIENT_TO_REQUEST_MDC,
27              BaseAuthorizationUtils.SECRET_TO_REQUEST_MDC);
28          try {
29              String resultStr = httpClient.execute(get, new BasicResponseHandler());
30              JSONObject resultJo = new JSONObject(resultStr);
31              JSONArray dataJa = resultJo.getJSONArray("data");
32              for (int i = 0; i < dataJa.length(); i++) {
33                  JSONObject itemJo = dataJa.getJSONObject(i);
34                  districtIdNameMap.put(itemJo.getInt("id"), itemJo.getString("name"));
35              }
36          } catch (Exception e) {
37              throw new RuntimeException("Init District List Error!", e);
38          }
39      }
```

```
41      public static String getCityName(int cityId) {
42          String name = cityIdNameMap.get(cityId);
43          if (name == null) {
44              name = "未知";
45          }
46          return name;
47      }
48
49      public static String getDistrictName(int districtId) {
50          String name = districtIdNameMap.get(districtId);
51          if (name == null) {
52              name = "未知";
53          }
54          return name;
55      }
56  }
```

编程直接实现缓存

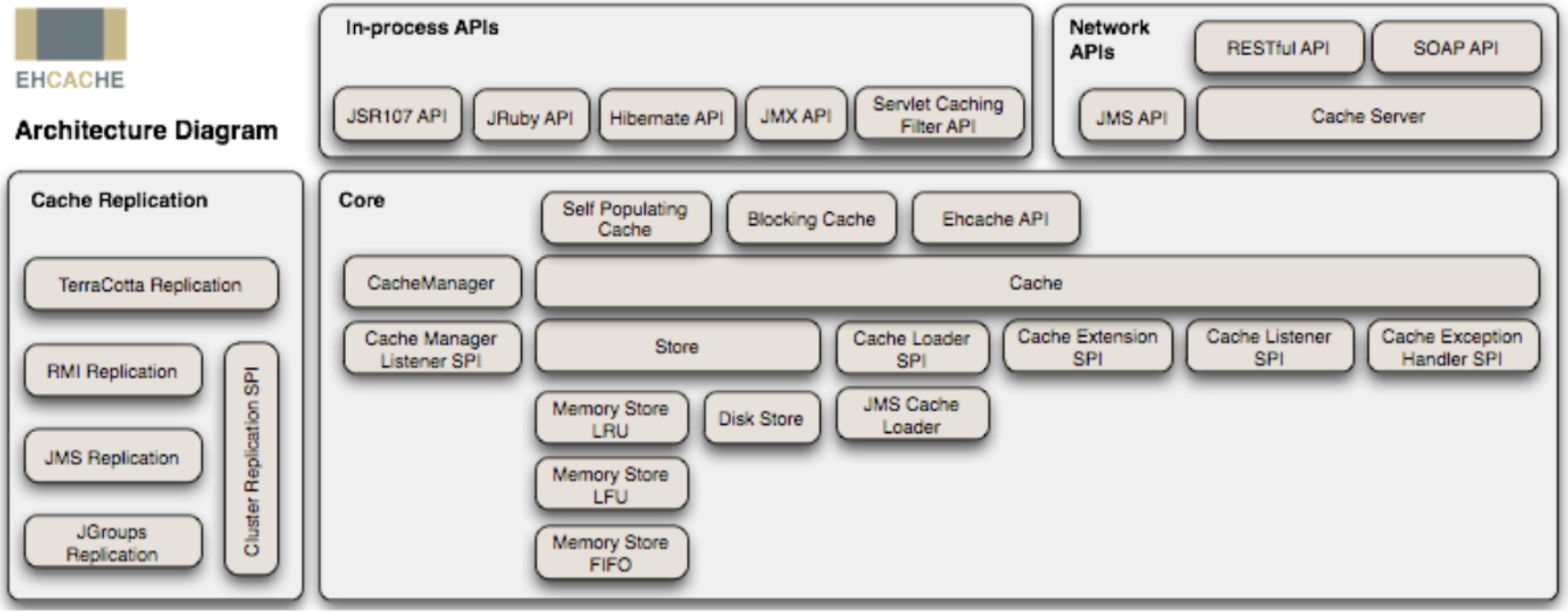
- 静态变量实现
  - 通过静态变量一次获取到缓存内存中，减少频繁的I/O读取，静态变量实现类间可共享，进程内可共享，缓存的实时性稍差。
  - 为了解决本地缓存数据的实时性问题，目前大量使用的是结合ZooKeeper的自动发现机制，实时变更本地静态变量缓存，如下图：



- 编程直接实现缓存，优点是能直接在heap区内读写，最快也最方便；缺点同样是受heap区域影响，缓存的数据量非常有限，同时缓存时间受GC影响。主要满足单机场景下的小数据量缓存需求，同时对缓存数据的变更无需太敏感感知，如上一般配置管理、基础静态数据等场景。

本地缓存 → 实现方法

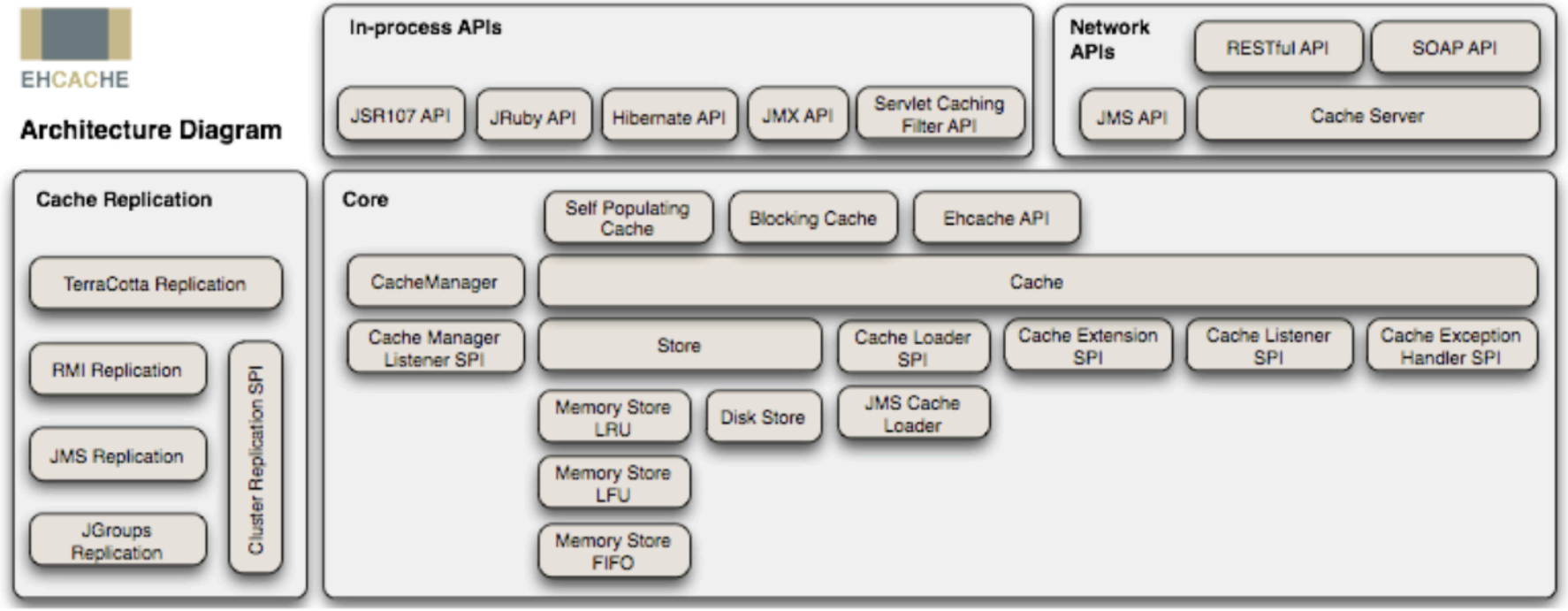
中间件Ehcache



- Ehcache是现在最流行的纯Java开源缓存框架，配置简单、结构清晰、功能强大，是一个轻量级的缓存实现，Hibernate里面集成了相关缓存功能。
- Ehcache的核心定义主要包括：
  - cache manager：缓存管理器，允许多实例
  - cache：缓存管理器内可以放置若干cache，存放数据的实质，所有cache都实现了Ehcache接口，这是一个真正使用的缓存实例；通过缓存管理器的模式，可以在单个应用中轻松隔离多个缓存实例，独立服务于不同业务场景需求，缓存数据物理隔离，同时需要时又可共享使用。

本地缓存 → 实现方法

中间件Ehcache



- element：单条缓存数据的组成单位。
- system of record（SOR）：可以取到真实数据的组件，可以是真正的业务逻辑、外部接口调用、存放真实数据的数据库等，缓存就是从SOR中读取或者写入到SOR中去的。

- 整个Ehcache提供了对JSR、JMX等的标准支持，能够较好的兼容和移植，同时对各类对象有较完善的监控管理机制。它的缓存介质涵盖堆内存（heap）、堆外内存（BigMemory商用版本支持）和磁盘，各介质可独立设置属性和策略。Ehcache最初是独立的本地缓存框架组件，在后期的发展中，结合Terracotta服务阵列模型，可以支持分布式缓存集群，主要有RMI、JGroups、JMS和Cache Server等传播方式进行节点间通信

中间件Ehcache

- Ehcache的使用还是相对简单便捷的，提供了完整的各类API接口。需要注意的是，虽然Ehcache支持磁盘的持久化，但是由于存在两级缓存介质，在一级内存中的缓存，如果没有主动的刷入磁盘持久化的话，在应用异常down机等情形下，依然会出现缓存数据丢失，为此可以根据需要将缓存刷到磁盘，将缓存条目刷到磁盘的操作可以通过cache.flush()方法来执行，需要注意的是，对于对象的磁盘写入，前提是要将对象进行序列化。
- 主要特性：
  - 快速，针对大型高并发系统场景，Ehcache的多线程机制有相应的优化改善。
  - 简单，很小的jar包，简单配置就可直接使用，单机场景下无需过多的其他服务依赖。
  - 支持多种的缓存策略，灵活。
  - 缓存数据有两级：内存和磁盘，与一般的本地内存缓存相比，有了磁盘的存储空间，将可以支持更大量的数据缓存需求。
  - 具有缓存和缓存管理器的侦听接口，能更简单方便的进行缓存实例的监控管理。
  - 支持多缓存管理器实例，以及一个实例的多个缓存区域。
- 注意：Ehcache的超时设置主要是针对整个cache实例设置整体的超时策略，而没有较好的处理针对单独的key的个性的超时设置，因此，在使用中要注意过期失效的缓存元素无法被GC回收，时间越长缓存越多，内存占用也就越大，内存泄露的概率也越大。

4.4
数据缓存

1、数据缓存的基本概念

2、本地缓存

3、分布式缓存

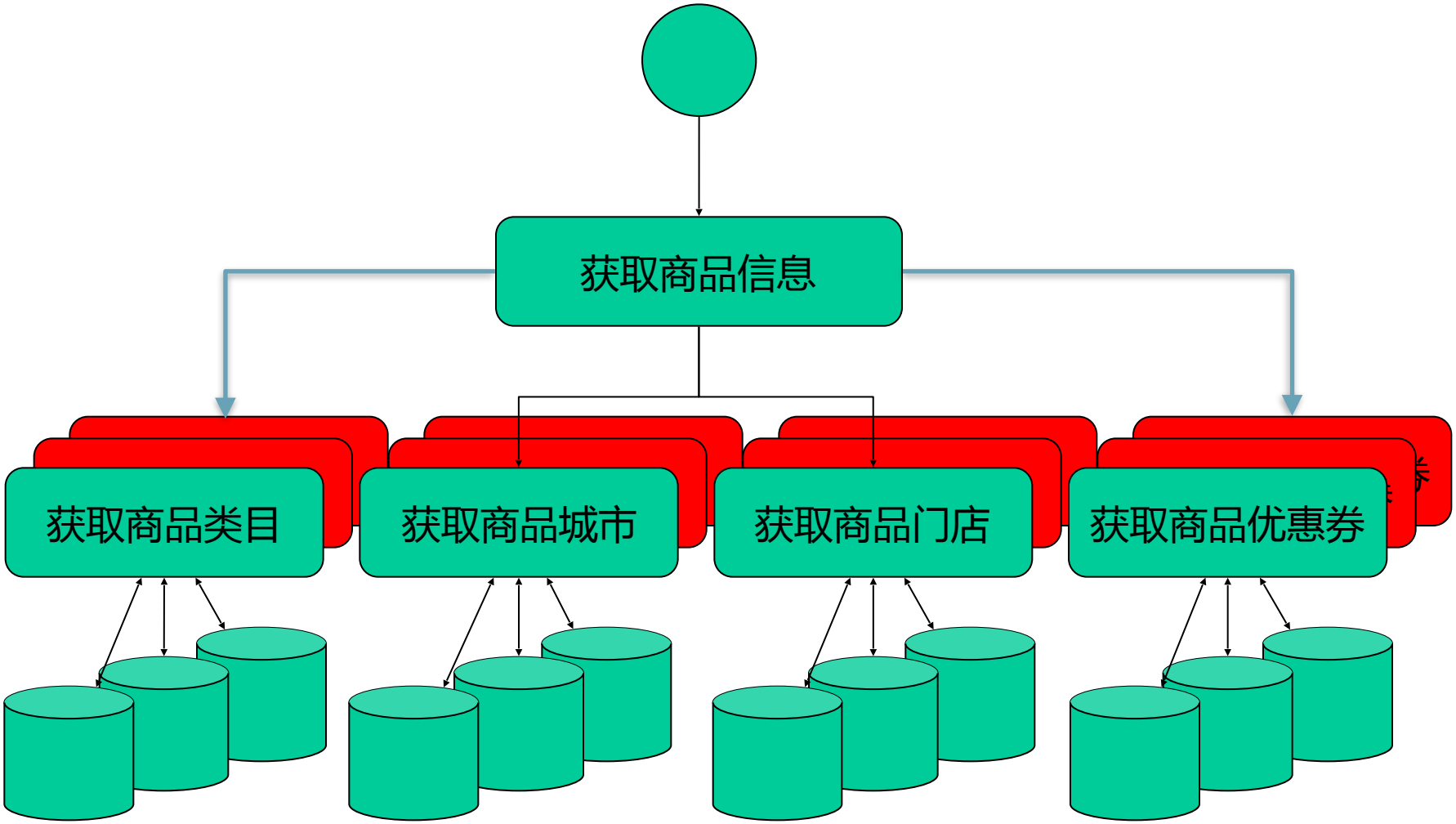
4、缓存问题讨论

4.4.3 分布式缓存

- 分布式缓存的关注点
- Redis

分布式缓存的关注点

分布式缓存基本形态的例子

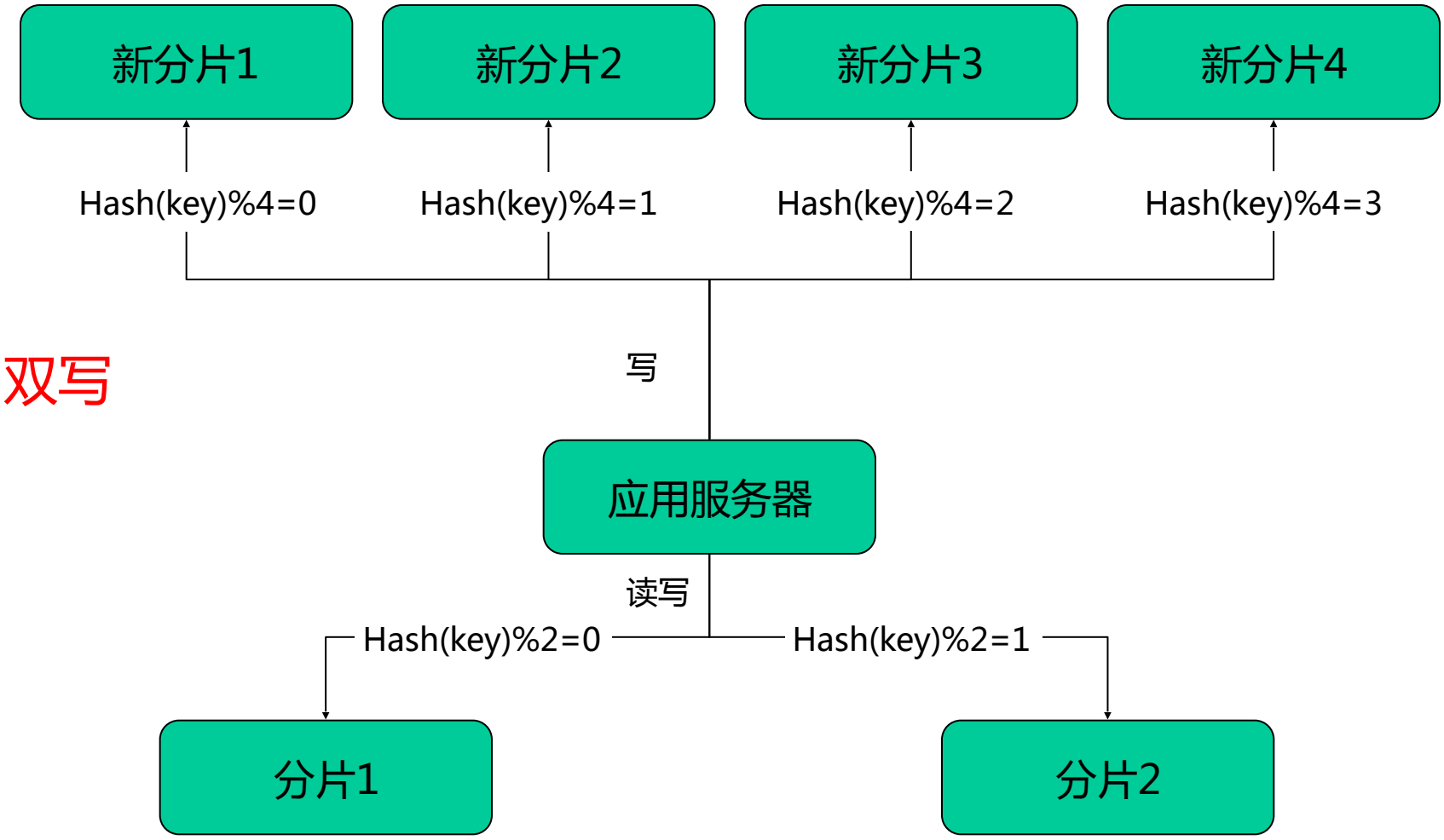


分布式缓存的分片

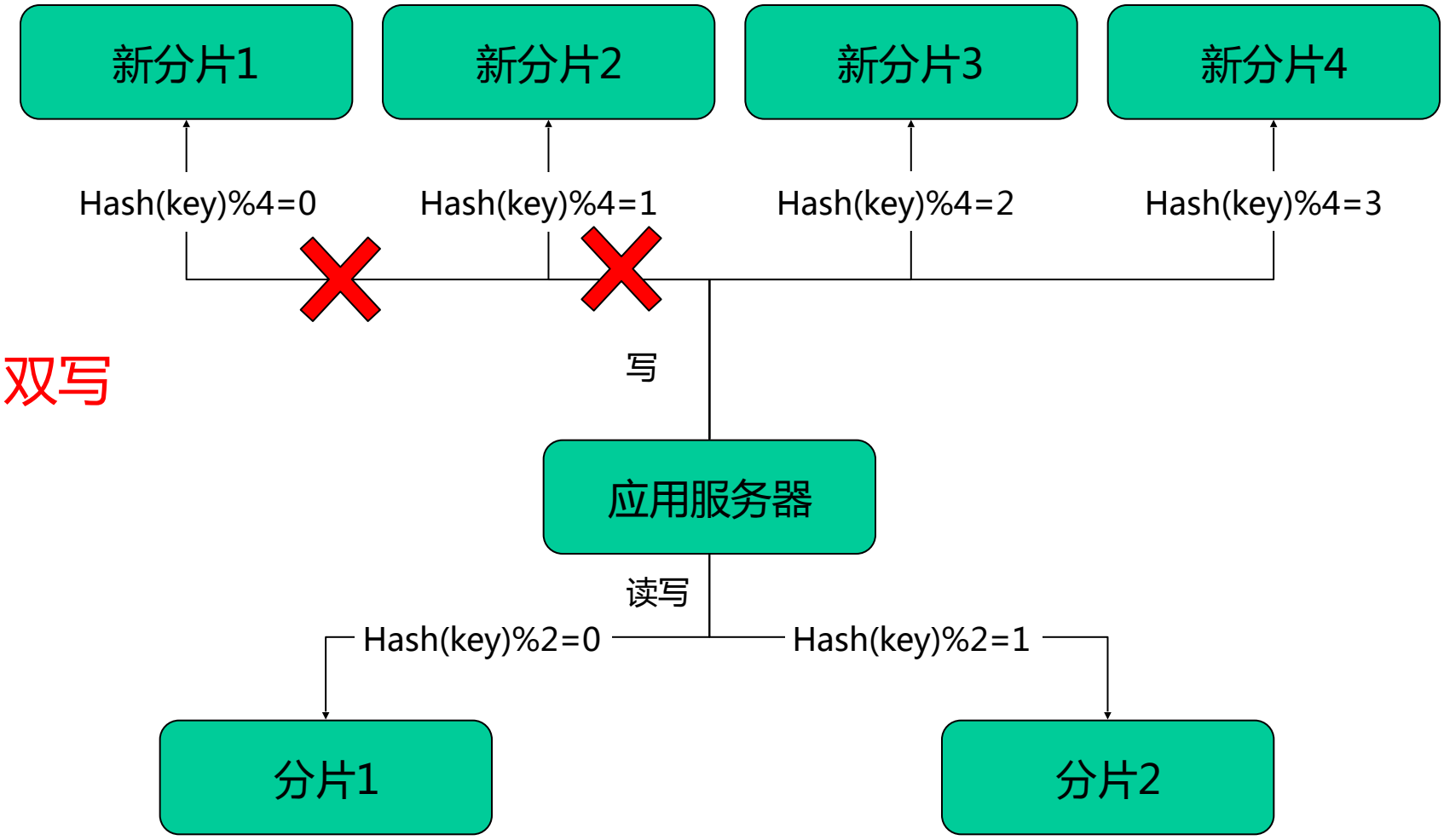
- 分片的基本原理与分库分表类似，不做赘述，参考分库分表。

分布式缓存的迁移-平滑迁移

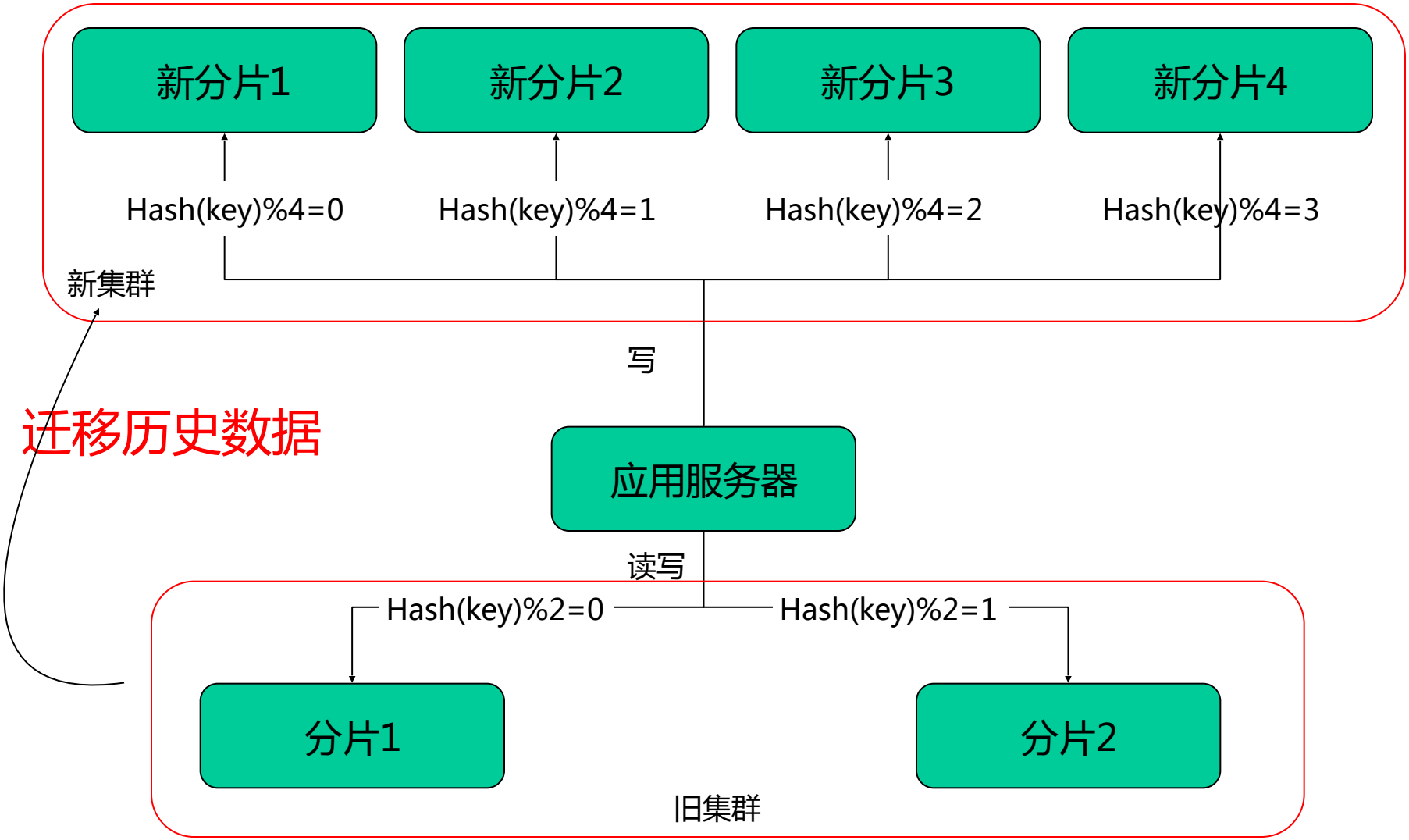
了解这个过程



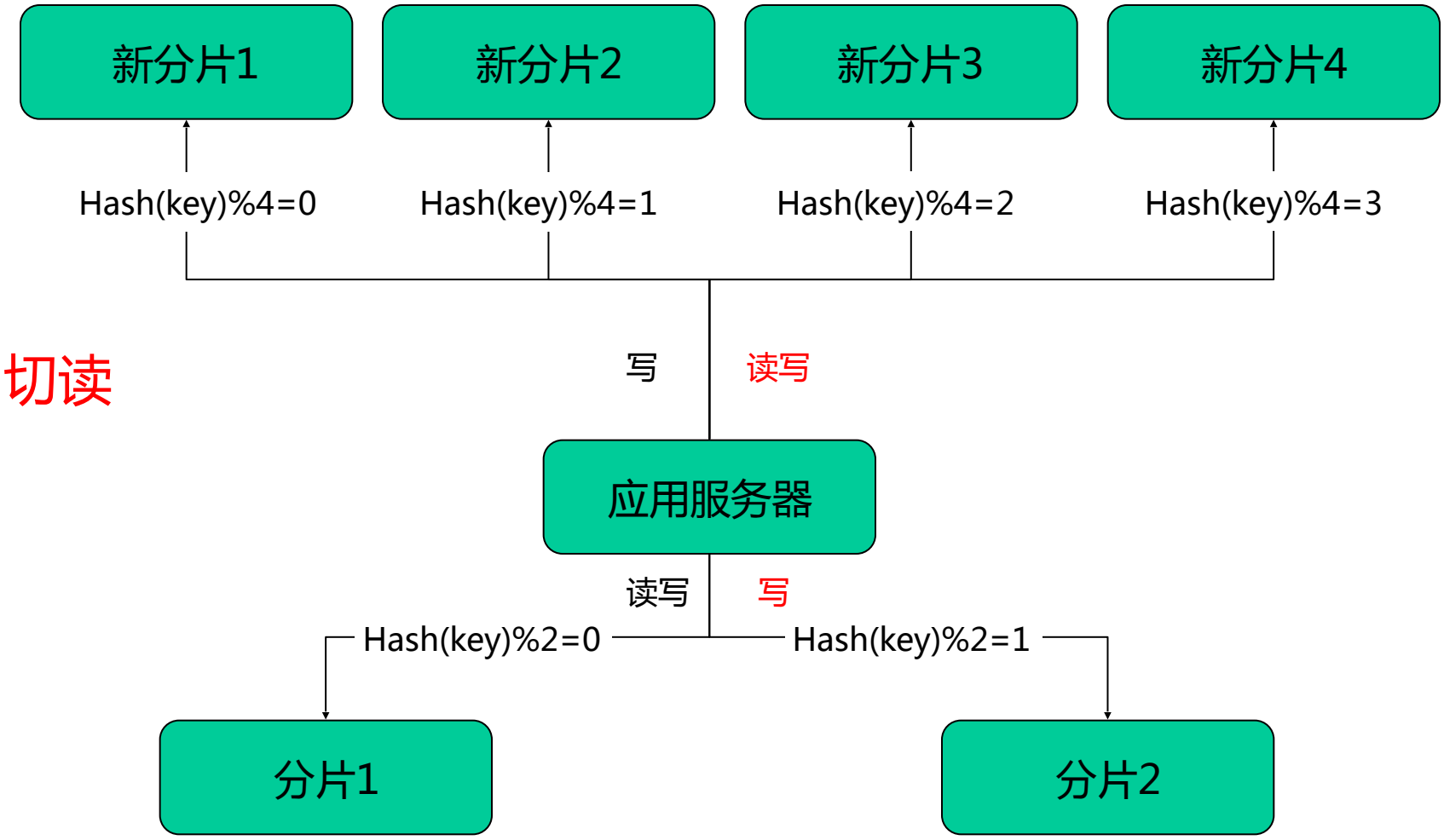
分布式缓存的迁移-平滑迁移



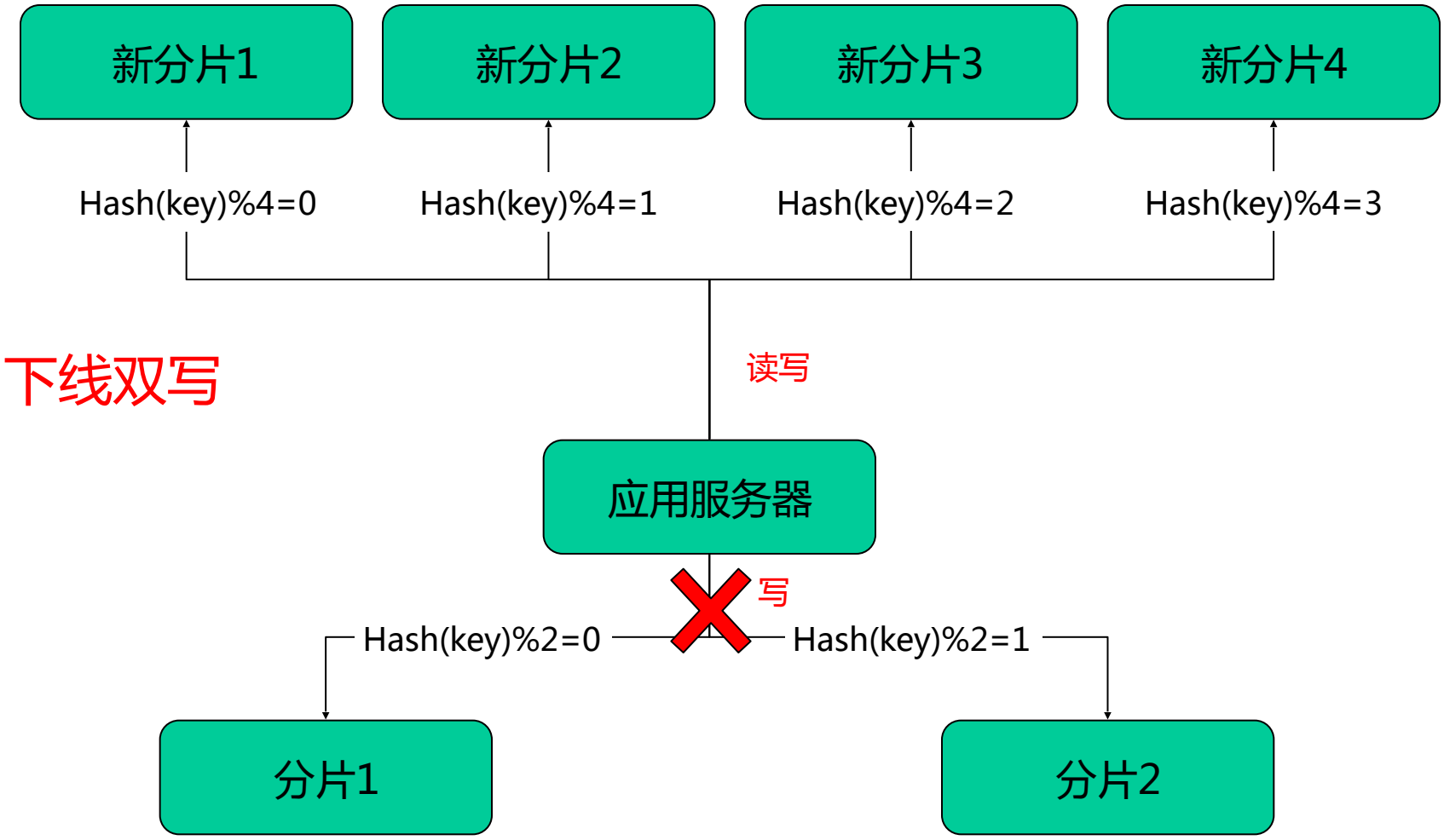
分布式缓存的迁移-平滑迁移



分布式缓存的迁移-平滑迁移



分布式缓存的迁移-平滑迁移



作业4

- 针对KWIC问题，我们要提供一个全球10亿用户可用的Web应用，请给出该系统的分布式架构设计方案
  - 用户可以从多种终端访问应用，e.g. 手机端、PC端
  - 可以通过网络上传待处理文件，文件可能超过10000行，历史数据需要存储并提供检索功能
  - 设计方案应考虑数据层的典型架构技术，实现存储的高可用和高性能
  - 应能够与作业3的计算层面架构形成有机整体
  - 截止日期：+2周

第4章

数据层的软件架构技术

Thanks for listening

涂志莹、苏统华
哈尔滨工业大学计算机学院
企业与服务计算研究中心