缓存使用的是内存资源，而内存资源是非常宝贵的，要用有限的服务器资源支撑更多的业务，就必须让那些访问频率不高的缓存删除掉，为新的缓存腾出内存空间。这一课时，我们一起来看一下，缓存失效有哪些策略。

页面置换算法

我们从一开始就提到，缓存技术在计算机系统设计中有着非常广泛的应用，对应到操作系统中，就是缓存页面的调度算法。

在操作系统中，文件的读取会先分配一定的页面空间，也就是我们说的 Page，使用页面的时候首先去查询空间是否有该页面的缓存，如果有的话，则直接拿出来；否则就先查询，页面空间没有满，就把新页面缓存起来，如果页面空间满了，就删除部分页面，方便新的页面插入。

在操作系统的页面空间中，对应淘汰旧页面的机制不同，所以会有不同页面调度方法，常见的有 FIFO、LRU、LFU 过期策略：

- **FIFO（First In First Out，先进先出）**，根据缓存被存储的时间，离当前最远的数据优先被淘汰；
- **LRU（Least Recently Used，最近最少使用）**，根据最近被使用的时间，离当前最远的数据优先被淘汰；
- **LFU（Least Frequently Used，最不经常使用）**，在一段时间内，缓存数据被使用次数最少的会被淘汰。

这三种策略也是经典的缓存淘汰策略，大部分缓存应用模型，都是基于这几种策略实现的。

内存淘汰策略

操作系统的页面置换算法，对应到分布式缓存中，就是缓存的内存淘汰策略，这里以 Redis 为例，进行分析。

当 Redis 节点分配的内存使用到达最大值以后，为了继续提供服务，Redis 会启动内存淘汰策略，以下的几种策略参考官方文档：

- noeviction，这是默认的策略，对于写请求会拒绝服务，直接返回错误，这种策略下可以保证数据不丢失；
- allkeys-lru，这种策略操作的范围是所有 key，使用 LRU 算法进行缓存淘汰；
- volatile-lru，这种策略操作的范围是设置了过期时间的 key，使用 LRU 算法进行淘汰；
-

allkeys-random，这种策略下操作的范围是所有 key，会进行随机淘汰数据；

- volatile-random，这种策略操作的范围是设置了过期时间的 key，会进行随机淘汰；

- volatile-ttl，这种策略操作的范围是设置了过期时间的 key，根据 key 的过期时间进行淘汰，越早过期的越优先被淘汰。

缓存过期策略

分布式缓存中的过期策略和内存淘汰策略又有一些不同，希望大家不要混淆，内存淘汰是缓存服务层面的操作，而过期策略定义的是具体缓存数据何时失效，下面一起来看一下。

我们都知道，Redis 是 key-value 数据库，可以设置缓存 key 的过期时间，过期策略就是指当 Redis 中缓存的 key 过期了，Redis 如何处理。

Redis 中过期策略通常有以下三种。

- 定时过期

这是最常见也是应用最多的策略，为每个设置过期时间的 key 都需要创建一个定时器，到过期时间就会立即清除。这种方式可以立即删除过期数据，避免浪费内存，但是需要耗费大量的 CPU 资源去处理过期的数据，可能影响缓存服务的性能。

- 惰性过期

可以类比懒加载的策略，这个就是懒过期，只有当访问一个 key 时，才会判断该 key 是否已过期，并且进行删除操作。这种方式可以节省 CPU 资源，但是可能会出现很多无效数据占用内存，极端情况下，缓存中出现大量的过期 key 无法被删除。

- 定期过期

这种方式是上面方案的整合，添加一个即将过期的缓存字典，每隔一定的时间，会扫描一定数量的 key，并清除其中已过期的 key。

合理的缓存配置，需要协调内存淘汰策略和过期策略，避免内存浪费，同时最大化缓存集群的吞吐量。另外，Redis 的缓存失效有一点特别关键，那就是如何避免大量主键在同一时间同时失效造成数据库压力过大的情况，对于这个问题在第 33 课时缓存穿透中有过描述，大家可以去扩展了解下。

实现一个 LRU 缓存

下面介绍一个高频的面试问题：如何实现一个 LRU 算法，该算法的实现很多同学都听过，但是不知道你还记不记得那句经典的格言，Talk is cheap，show me the code。很多人在写代码时一说就懂，一写就错，特别在面试时，常常要求你白板编程，脱离了 IDE 的帮助，更容易出现错误，所以我建议大家动手去实现一下。

在 Java 语言中实现 LUR 缓存，可以直接应用内置的 LinkedHashMap，重写对应的 removeEldestEntry() 方法，代码如下：

```
public class LinkedHashMapExtend extends LinkedHashMap {
    private int cacheSize;
```

```
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">LinkedHashMapExtend</span><span class="hljs-params">(<span
class="hljs-keyword">int</span> cacheSize)</span></span>{
    <span class="hljs-keyword">super</span>();
    <span class="hljs-keyword">this</span>.cacheSize=cacheSize;
}
<span class="hljs-meta">@Override</span>
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">boolean</span> <span class="hljs-
title">removeEldestEntry</span><span class="hljs-params">(Map.Entry eldest)</span>
</span>{
<span class="hljs-comment">//重写移除逻辑 </span>
    <span class="hljs-keyword">if</span>(size()&amp;gt;cacheSize){
        <span class="hljs-keyword">return</span> <span class="hljs-
keyword">true</span>;
    }
    <span class="hljs-keyword">return</span> <span class="hljs-
keyword">false</span>;
}
```

```
}
```

为什么重写 LinkedHashMap 可以实现 LRU 缓存呢？

对于这个问题，我建议你可以查看一下 LinkedHashMap 的源码实现，在原生的 removeEldestEntry 实现中，默认返回了 false，也就是永远不会移除最“早”的缓存数据，只要扩展这个条件，缓存满了移除最早的数据，是不是就实现了一个 LRU 策略？

在面试中，单纯使用 LinkedHashMap 实现是不够的，还会要求你使用原生的 Map 和双向链表来实现。下面我简单实现了一个参考代码，这道题目在 Leetcode 上的编号是 146，也是剑指 offer 里的一道经典题，大家可以去力扣网站提交代码试一下。

```
import java.util.HashMap;
public class LRUCache {
    private int cacheSize;
    private int currentSize;
    private CacheNode head;
    private CacheNode tail;
    private HashMap<Integer,CacheNode> nodes;
```

```
class CacheNode{
    CacheNode prev;
    CacheNode next;
    int key;
    int value;
}

public LRUCache(int cacheSize){
    cacheSize=cacheSize;
    currentSize=0;
    nodes=new HashMap<>(cacheSize);

}

public void set(Integer key,Integer value){
    if(nodes.get(key)==null){ //添加新元素
        CacheNode node=new CacheNode();
        node.key=key;
        node.value=value;
        nodes.put(key,node);
        //移动到表头
        moveToHead(node);
        //进行lru操作
        if(currentSize>cacheSize)
            removeTail();
        else
            currentSize++;

    }else{//更新元素值
        CacheNode node=nodes.get(key);
        //移动到表头
        moveToHead(node);
        node.value=value;
    }

}

private void removeTail() {
    if(tail!=null){
        nodes.remove(tail.key);
        if(tail.prev!=null)  tail.prev.next=null;
        tail=tail.prev;
```

```
        }
    }

    private void moveToHead(CacheNode node){
        //链表中间的元素
        if(node.prev!=null){
            node.prev.next=node.next;
        }
        if(node.next!=null){
            node.next.prev=node.prev;
        }

        //移动到表头
        node.prev=null;
        if(head==null){
            head=node;
        }else{
            node.next=head;
            head.prev=node;
        }

        head=node;
        //更新tail
        //node就是尾部元素
        if(tail==node){
            //下移一位
            tail=tail.prev;
        }

        //缓存里就一个元素
        if(tail==null){
          tail=node;
        }
    }

    public int get(int key){
        if(nodes.get(key)!=null){
            CacheNode node=nodes.get(key);
            moveToHead(node);
            return node.value;
        }
        return 0;
    }
}
```

总结

这一课时的内容主要介绍了缓存的几种失效策略，并且分享了一个面试中的高频问题：LRU 缓存实现。

缓存过期的策略来自操作系统，在我们的专栏中，对很多知识的展开都来自计算机原理、网络原理等底层技术，也从一个侧面反映了计算机基础知识的重要性。

精选评论