在业务开发中，缓存服务和其他数据服务一样，需要满足高可用性，而高可用最常用的手段就是**集群扩展。**

缓存的集群高可用

目前 Redis 流行的集群方案有 官方 Cluster 方案、twemproxy 代理方案、哨兵模式、Codis 等方案，关于这几种方案的具体应用，我们在下一课时将详细展开讲解。

缓存服务从单点扩展到集群以后，势必会产生缓存数据的分发问题，假设我们的缓存服务器有 3 台，每台缓存的数据是不相同的，那么我们在更新缓存时，该放置在哪台机器上呢？根据 key 获取缓存时，该从哪台服务器上获取？这就涉及缓存的**负载均衡策略。**

关于缓存集群高可用的配置方式，有数据同步和不同步之分。在数据同步的方案下，所有节点之间数据都是一样的，不同节点互为副本，这种方式不需要关心缓存数据的分发，实现了缓存集群的最大可用，但是由于冗余了多份缓存数据，会造成比较多的服务器资源浪费；另外一方面，在更新缓存数据时，还要考虑不同节点之间的一致性。

数据不同步的方案，就是每个缓存节点存储的数据不同，在缓存读写时使用一定的策略进行分发。在实际开发中，大部分都是应用数据不同步的方案，如果需要冗余数据，则可以通过缓存集群主从同步实现。

不同路由方案的扩容问题

在第 22 课时讲解数据库分库分表时，我们分析了数据库分库分表扩容的问题，分库分表以后，当存储节点发生增加或减少时，合理的配置分表策略，可以使得数据迁移最小。

其实不只是数据库，缓存集群也有一样的问题。下面来看一下几种负载均衡策略，以及对应的优缺点。

哈希取模路由

最常见的方式是对缓存数据进行哈希，典型的操作就是通过对缓存 hash（缓存 Key）/ 节点数量。

假设我们有 5 台缓存服务器，伪代码如下：

```
//获取缓存服务器下标
public Integer getRoute(String key){
int cacheIndex = key.hashcode() % 5;
return cacheIndex;
}
```

哈希取模的方式，适合对固定数量的缓存集群进行路由，但是对横向扩展不友好。如果缓存机器数量发生变更过，比如从 5 台服务器调整为 10 台服务器，原来的缓存数据无法分配到正确机器，就会出现路由不正确，从而业务请求直接落到数据库上。

一致性哈希

在负载均衡策略中，可以应用一致性哈希，减少节点扩展时的数据失效或者迁移的情况。维基百科对一致性哈希是这么定义的：

> 一致性哈希是一种特殊的哈希算法。在使用一致性哈希算法后，哈希表槽位数（大小）的改变平均只需要对 K/n 个关键字重新映射，其中 K 是关键字的数量，n 是槽位数量。然而在传统的哈希表中，添加或删除一个槽位几乎需要对所有关键字进行重新映射。

一致性哈希通过一个哈希环实现，Hash 环的基本思路是获取所有的服务器节点 hash 值，然后获取 key 的 hash，与节点的 hash 进行对比，找出顺时针最近的节点进行存储和读取。

以电商中的商品数据为例，假设我们有 4 台缓存服务器：

- A 服务器，地址 hash 结果是 100
- B 服务器，地址 hash 结果是 200
- C 服务器，地址 hash 结果是 300
- D 服务器，地址 hash 结果是 400

现在有某条数据的 Key 进行哈希操作，得到 200，则存储在 B 服务器；某条数据的 Key 进行哈希操作，得到 260，则存储在 C 服务器；某条数据的 Key 进行哈希操作，得到 500，则存储在 A 服务器。

一致性哈希算法在扩展时，只需要迁移少量的数据就可以。例如，我们刚才的例子中，如果 D 服务器下线，原先路由到 D 服务器的数据，只要顺时针迁移到 A 服务器就可以，其他服务器不受影响，我们只需要移动一台机器的数据即可。

一致性哈希虽然对扩容和缩容友好，但是存在另外一个问题，就很容易出现数据倾斜。

相信你已经考虑到了，假设我们有 A、B、C 一直到 J 服务器，总共 10 台，组成一个哈希环。如果从 F 服务器一直到 J 服务器的 5 个节点宕机，那么这 5 台服务器原来的访问，都会被转移到服务器 A 之上，服务器的流量可能是原来的 5 倍或者更高，直到把服务器 A 打爆，这时候流量继续转移到 B 服务器，就出现我们在第 34 课时提到的**缓存雪崩**。

那么数据倾斜是如何解决的呢? 一个方案就是添加虚拟节点，对服务器节点也进行哈希操作，在整个哈希环上，均匀添加若干个节点。比如 a1 和 a2 都属于 A 节点，b1、b2 都属于 B 节点，这样在哈希时可以平衡各个节点的数据。

另外，在面试中，面试官可能会要求你实现一致性哈希算法。以 Java 为例，可以应用 TreeMap 这个数据结构。

TreeMap 基于红黑树实现，元素默认按照 keys 的自然排序排列，对外开放了一个 tailMap(K fromKey) 方法，该方法可以返回比 fromKey 顺序的下一个节点，大大简化了一致性哈希的实现。这里我就不添加代码了，感兴趣的同学可以去动手模拟实现一下。

总结

这一课时的内容，和你分享了应用缓存集群的知识点，包括集群下的高可用，以及哈希取模和一致性哈希的负载均衡策略。

一致性哈希算法的应用，主要是考虑到分布式系统每个节点都有可能失效，并且新的节点很可能动态地增加进来的情况，如何保证当系统的节点数目发生变化的时候，我们的系统仍然能够对外提供良好的服务。

负载均衡在分布式系统设计中是非常重要的一部分，今天主要关注的是数据路由方案，除了数据路由，负载均衡在 API 网关、分布式服务调用中也非常关键。在服务调用中常用的负载均衡策略还包括轮训、随机，根据响应时间判断等。在你的工作中，有哪些场景用到了负载均衡，又是如何进行应用的呢？欢迎留言进行分享。

精选评论

***样：**

> nginx中基于客户的IP还有基于客户的请求头埋点的方式来决定请求的服务器方式