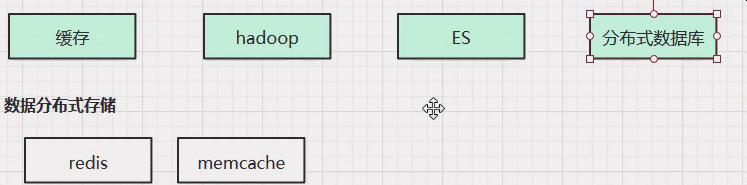
一致性哈希Hash算法

# 一致性Hash算法的应用领域

用于有关**数据分布式存储**的场景。

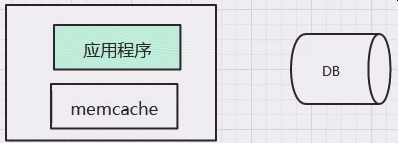
缓存、Hadoop、搜索引擎如ES、分布式数据库如分库分表。



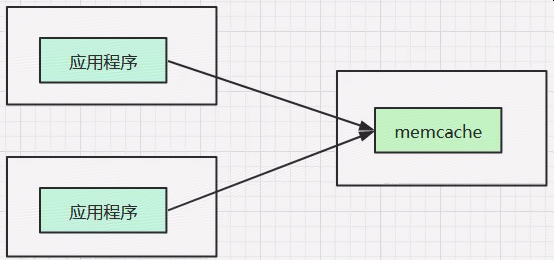
# 分布式缓存

常用的缓存中间件：**Redis和MemCached**。最常用的两个。

## 本地缓存：应用程序与缓存部署在一起，称作本地缓存；



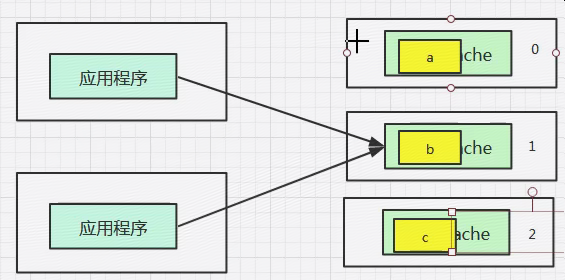
## 远程分布式缓存



## 大型高并发系统

高并发：20W/s

海量数据



# 传统的取模方式

例如10条数据，3个节点，如果按照取模的方式，那就是

node a: 0,3,6,9

node b: 1,4,7

node c: 2,5,8

当增加一个节点的时候，数据分布就变更为

node a:0,4,8

node b:1,5,9

node c: 2,6

node d: 3,7

总结：数据3,4,5,6,7,8,9在增加节点的时候，都需要做搬迁，成本太高。

扩容：3台机器变成4台机器，hash/3变成hash/4；将会导致**75%**的数据无法命中，导致雪崩。

计算过程：3—>4，在0-11范围内，只有0、1、2命中，则其他都命不中，3/(3\*4)=1/4即只有1/4的数据能够命中，而75%的数据无法命中。

规律：如果机器一台一台地添加，原来m台机器，扩容1台，则会导致m/(m+1)的数据无法命中。计算公式：无法命中概率**1- m/(m\*(m+1))=m/(m+1)**。导致缓存雪崩。

解决方法：**在凌晨扩容，需要数据预热**。

## 一致性哈希方式

核心思想：**对节点和数据，都做一次哈希运算**，然后比较节点和数据的哈希值，按照顺时针方向取哈希值hash最相近的节点做为**存放节点**。这样就保证当节点增加或者减少的时候，影响的数据最少。

### hash值是个非负整数，非负整数的值范围做成一个环；

### 对集群的节点的某个属性（比如节点名）求hash值，放到环上；

**《对节点属性求hash值，作为临界点》**。

### 对数据key求hash值，也放到环上，按照顺时针方向找到离它最近的节点，放到它上面。

一致性hash环存在问题：

### 1不能保证数据均衡分布；

### 加一个节点不能均衡减轻集群其他节点的压力。

示例：

用简单的字符串的ascii码做哈希key：

十条数据，算出各自的哈希值

**0：192**

**1：196**

**2：200**

3：204

4：208

5：212

6：216

7：220

8：224

9：228

有三个节点，算出各自的哈希值

node a: 203

node g: 209

node z: 228

这个时候比较两者的哈希值，**如果大于228，就归到前面的203**，相当于整个哈希值就是一个环，对应的映射结果：

node a: 0,1,2

node g: 3,4

node z: 5,6,7,8,9

这个时候加入node n, 就可以算出node n的哈希值：

node n: 216

这个时候对应的数据就会做迁移：

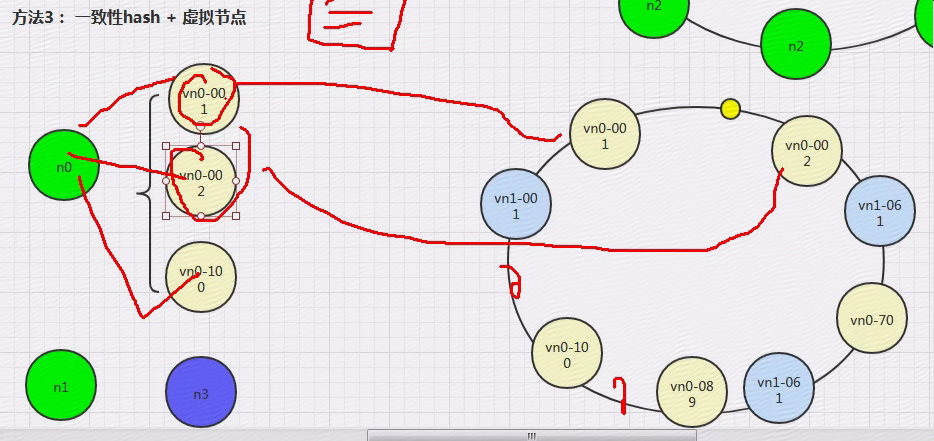
node a: 0,1,2

node g: 3,4

**node n: 5,6**

**node z: 7,8,9**

## 这个时候比较两者的哈希值，如果大于228，就归到前面的203，相当于整个哈希值就是一一致性Hash+虚拟节点



开设虚拟节点，一般情况下，节点数越多越好。

## “一致性”的理解：新加入的节点不影响原来已有节点保存的数据，即保持原有节点数据保存的一致性。

**影响越小越好。**