目录

[1. STL中unordered\_Map的实现 2](#_Toc68636030)

[2. vector与list 2](#_Toc68636031)

[3. Vector动态内存分配 2](#_Toc68636032)

[4. Vector如何扩容，为什么不用2倍 3](#_Toc68636033)

[5. Vector中reserve和resize的区别、内存不足 3](#_Toc68636034)

[6. Vector和list的区别，优缺点 3](#_Toc68636035)

[7. 如何在共享内存上使用stl标准库 3](#_Toc68636036)

[8. STL sort()底层实现 3](#_Toc68636037)

[9. map和hash\_map的区别 3](#_Toc68636038)

[10. Stl的配置器 3](#_Toc68636039)

[11. map和set底层实现及区别 3](#_Toc68636040)

[12. stl六大组件 3](#_Toc68636041)

[13. IEG光子 3](#_Toc68636042)

[14. 源码剖析unordered\_map解决hash冲突的 过程（与哈希表的扩容机制） 3](#_Toc68636043)

[15. 一致性哈希 3](#_Toc68636044)

[16. map的实现原理 3](#_Toc68636045)

[17. 红黑树特点，于平衡二叉树的区别，跟一般搜索树的区别（要在RB\_Tree上及产能和红非叶子节点的删除操作，左旋右旋，旋转次数比较 3](#_Toc68636046)

[https://blog.csdn.net/z702143700/article/details/49079107） 3](#_Toc68636047)

[18. 红黑树的结构以及如何调整树结构的 3](#_Toc68636048)

[19. Vector和set的区别 4](#_Toc68636049)

[20. Vector底层实现，扩容，删除（插入第八个元素会发生什么），push\_back和emplace\_back 4](#_Toc68636050)

[21. gdb调试打断点命令是什么？ 4](#_Toc68636051)

[22. 4](#_Toc68636052)

[23. 4](#_Toc68636053)

[24. 4](#_Toc68636054)

[25. 4](#_Toc68636055)

[26. 4](#_Toc68636056)

[27. 4](#_Toc68636057)

[28. 4](#_Toc68636058)

[29. STL sort()底层实现 4](#_Toc68636059)

1. vector与list

vector

vector与数组类似，拥有一段连续的内存空间，并且起始地址不变。便于随机访问，时间复杂度为O(1)，但因为内存空间是连续的，所以在进入插入和删除操作时，会造成内存块的拷贝，时间复杂度为O(n)

list

list底层是由双向链表实现的，因此内存空间不是连续的。根据链表的实现原理，list查询效率较低，时间复杂度为O(n)，但插入和删除效率较高。只需要在插入的地方更改指针的指向即可，不用移动数据。

迭代器支持不同

异：vector中，iterator支持“+”、“+=”，“<”等操作。而list中则不支持

同：vector<int>::iterator和list<int>::iterator都重载了“++”操作。

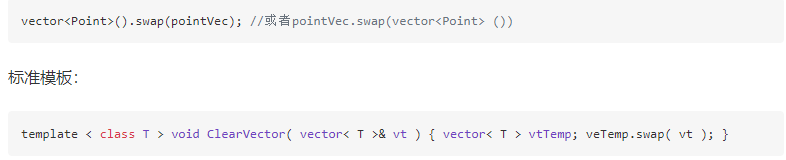
1. Vector动态内存分配

vector其中一个特点：内存空间只会增长，不会减小：为了支持快速的随机访问，vector容器的元素以连续方式存放，每一个元素都紧挨着前一个元素存储。当vector添加一个元素时，为了满足连续存放的特性，都需要重新分配空间、拷贝元素、撤销旧空间，这样性能难以接受。因此STL实现者在对vector进行内存分配时，其实际分配的容量要比当前所需的空间多一些。就是说，vector容器预留了一些额外的存储区，用于存放新添加的数据。这样不必每个新元素重新分配整个容器的内存空间。

vector的内存释放

由于vector的内存占用只增不减，比如你首先分配了10000个字节，然后erase了前面9999个，留下一个有效元素，但是内存占用仍为10000个。所有内存空间时在vector析构时候才能被系统回收。empty()用来检测容器是否为空，clear可以清空所有元素。但是即时clear，vector所占用的内存空间依然如故，无法保证内存的回收。

所以如果需要空间动态缩小，可以考虑使用deque。如果vector，可以用swap()来帮助你释放内存。具体实现



swap()是交换函数，使vector离开其自身的作用域，从而强制释放vector所占的内存空间，总而言之，释放vector内存的最简单方法使vector<point>().swap(pointVec)。如果pointVec是一个类成员，不能把该函数写入类的析构函数当中，否则会可能出现重复释放内存的错误。

其它情况释放内存：

如果vector中存放的是指针，那么当vector销毁时，这些指针指向对象不会被销毁，那么内存就不会被释放。

考虑使用智能指针

1. vector在类中声明是有个二义性的缺点

在vector<vector<int>> vec({})这种初始化时，在类中会有二义性，不知道你是在声明成员函数还是类成员

所以要变成左值

1. 如何在共享内存上使用stl标准库
2. STL sort()底层实现

数据量大时采用快速排序QuickSort，分段递归排序。一旦分段后数据量小于某种阈值，为避免QuickSort的递归调用带来过来的额外开销，就改用插入排序insertion Sort。如果递归层数过深，还会改用堆排序Heap sort.

1. map和hash\_map的区别

hash\_map的hash函数以及解决地址冲突等都要耗时间，而且众所周知hash表是以空间换时间的，因而hash\_map的内存消耗肯定要大，一般情况下，如果记录非常大，考虑hash\_map，查找效率会高很多，如果要考虑内存消耗，则要谨慎使用hash\_map。

简单理解：

Map集合的特点：

1. map集合一次存储两个对象，一个键对象，一个值对象
2. 键对象在集合中是唯一的，可以通过键来查找值

HashMap特点：

1. 使用哈希算法对键去重复、效率高，但无序
2. HashMap是Map接口的主要实现类

std::map内部是排序的，内部使用的是红黑树实现，不管是增加还是查找时间复杂度O(logn)，而std::unordered\_map其增加和查询的时间复杂度才是O(1)。它提供了类似map的方法

1. Stl的配置器
2. map和set底层实现及区别
3. stl六大组件（\*）
4. 什么是配接器
5. vector内存如何分配？
6. 分配器对于大于 128bytes的空间如何分配？
7. 内存池不够了怎么办？
8. 一致性哈希

分布式系统中对象与节点的映射关系，传统方案是使用对象的哈希值，对节点个数取模，再映射到相应编号的节点，这种方案在节点个数变动时，绝大多数对象的映射关系会失效而需要迁移；而一致性哈希算法中，当节点个数变动时，映射关系失效的对象非常少，迁移成本也非常小。

传统哈希（硬哈希）

分布式系统中，假设有 n 个节点，传统方案使用 mod(key, n) 映射数据和节点。  
当扩容或缩容时(哪怕只是增减1个节点)，映射关系变为 mod(key, n+1) / mod(key, n-1)，绝大多数数据的映射关系都会失效。

一致性哈希（Consistent Hashing）

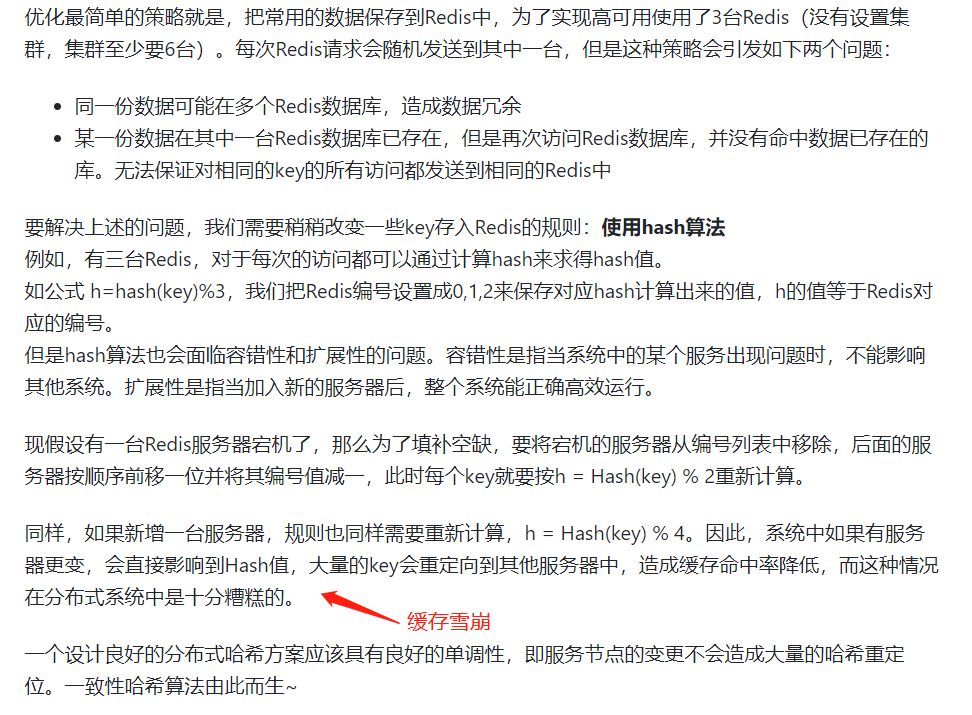
均衡性(Balance)：将关键字的哈希地址均匀地分布在地址空间中，使地址空间得到充分利用，这是设计哈希的一个基本特性。

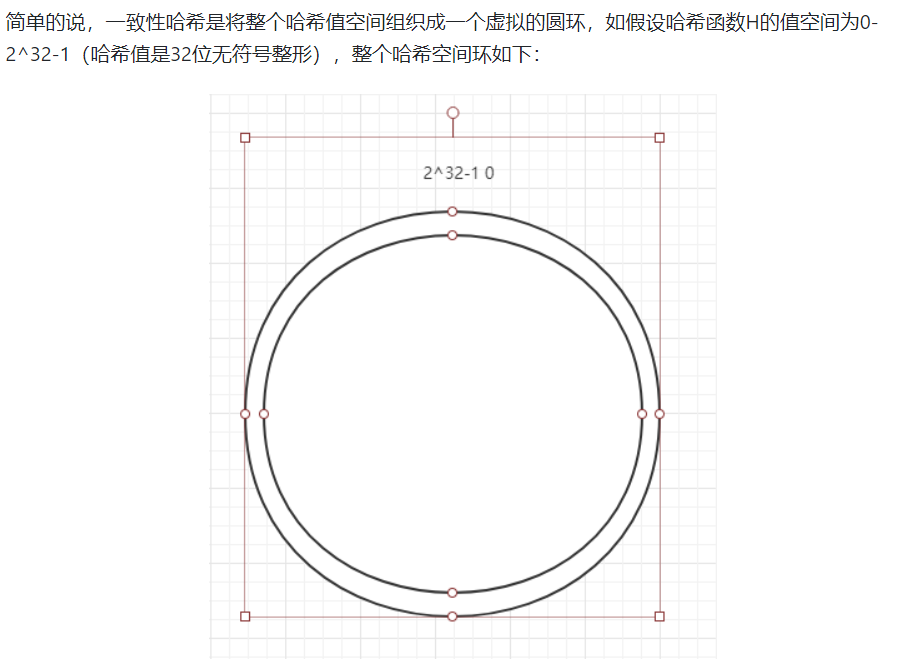
单调性(Monotonicity): 单调性是指当地址空间增大时，通过哈希函数所得到的关键字的哈希地址也能映射的新的地址空间，而不是仅限于原先的地址空间。或等地址空间减少时，也是只能映射到有效的地址空间中。简单的哈希函数往往不能满足此性质。

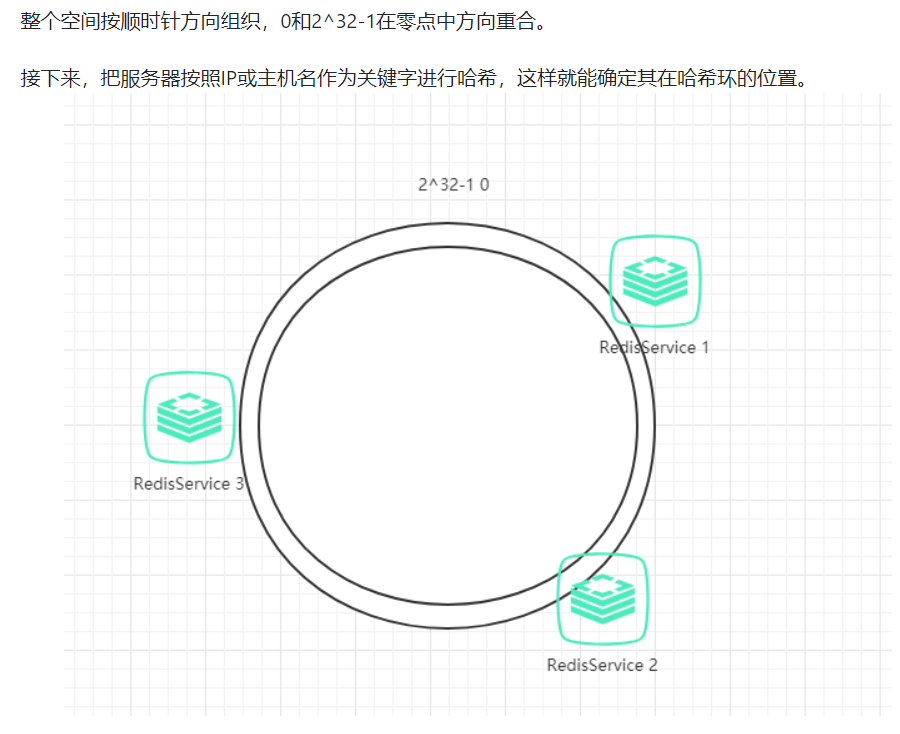
一致哈希 是一种特殊的哈希算法。在使用一致哈希算法后，哈希表槽位数（大小）的改变平均只需要对 K/n 个关键字重新映射，其中K是关键字的数量， n是槽位数量。然而在传统的哈希表中，添加或删除一个槽位的几乎需要对所有关键字进行重新映射。

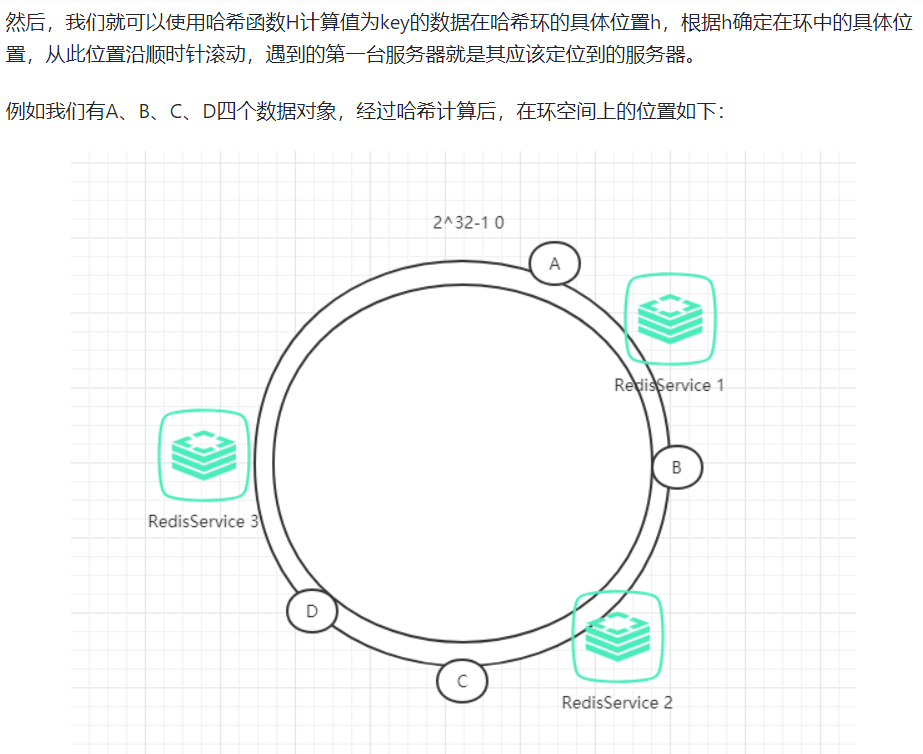
一致性哈希：

故事前景：





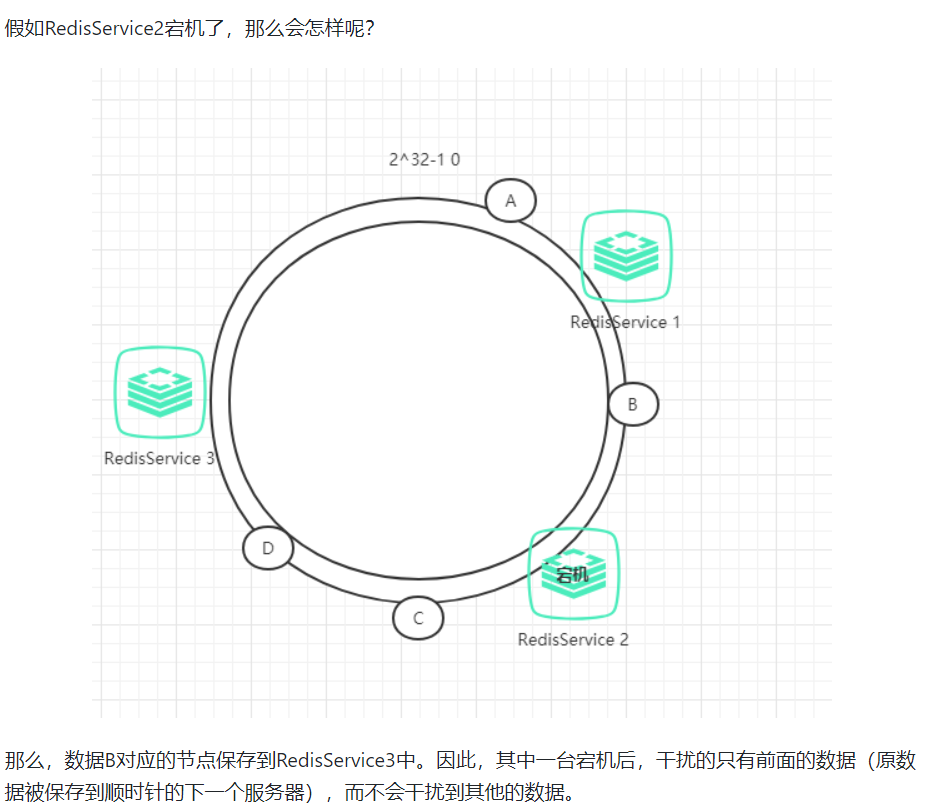




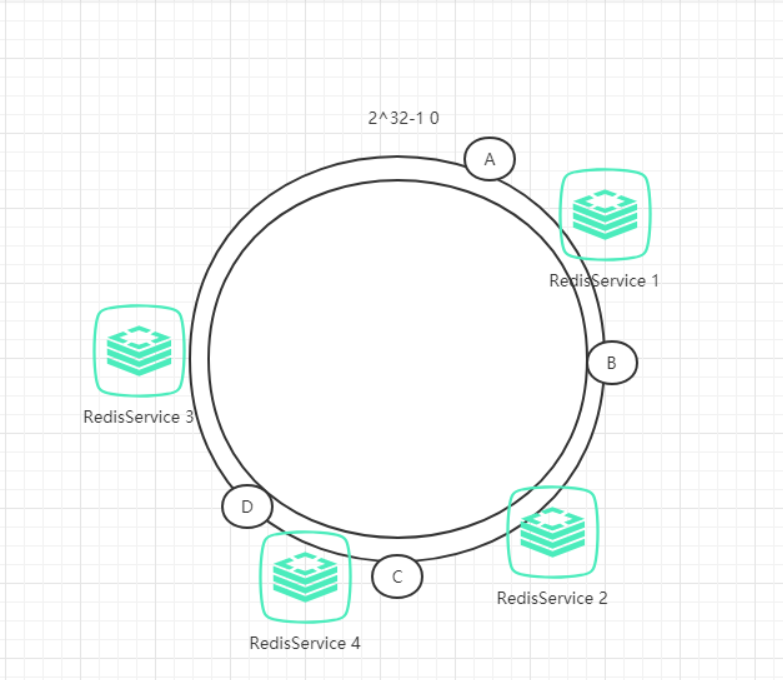
根据一致性哈希算法，数据A会被定为到Server 1上，数据B被定为到Server 2上，而C、D被定为到Server 3上。

容错性和扩展性：

容错性：



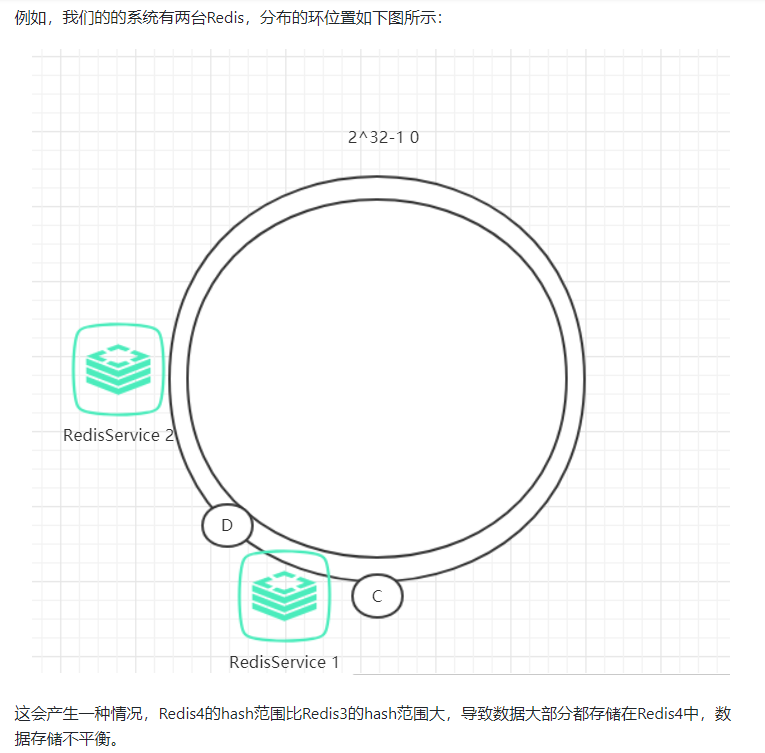
扩展性：



原本数据C是保存到Redis3中，但由于增加了Redis4，数据C被保存到Redis4中。干扰的也只有Redis3而已，其他数据不会受到影响。

因此，一致性哈希算法对于节点的增减都只需重定位换空间的一小部分即可，具有较好的容错性和可扩展性

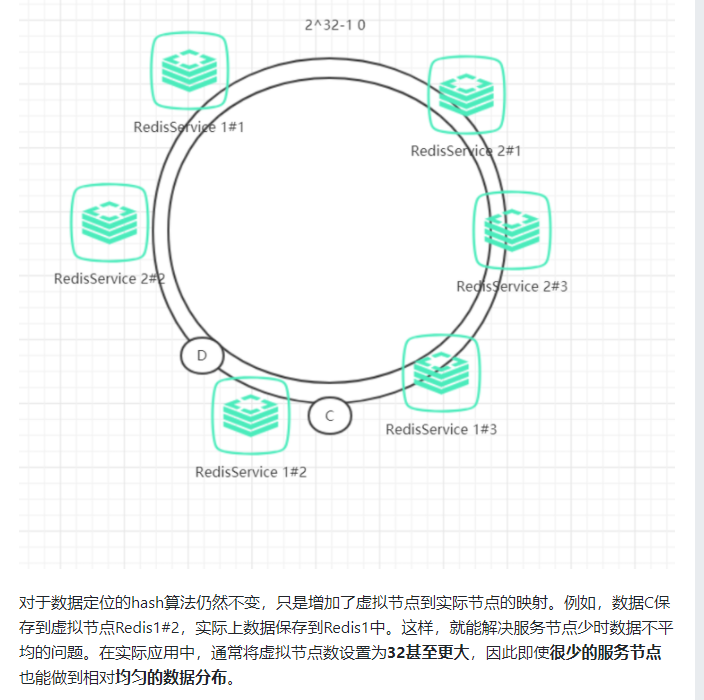
一致性哈希偏斜：



解决方法1：服务器尽可能的多

2：引入虚拟节点机制。即对每个节点计算多个哈希值，每个计算结果位置都放置在对应节点中，这些节点**称为虚拟节点**。

具体做法可以在服务器IP或主机名的后面增加编号来实现，例如上面的情况，可以为每个服务节点增加三个虚拟节点，于是可以分为 RedisService1#1、 RedisService1#2、 RedisService1#3、 RedisService2#1、 RedisService2#2、 RedisService2#3，具体位置如下图所示：



目前一致性哈希算法基本成为了**分布式系统组件的标准配置**，因此，我们十分有必要了解该算法。

但是因为宕机后需要重新从数据库读取，会导致此时缓存命中率下降及db压力增加。  
也无法避免单一热点问题。某一数据被海量请求，不论怎么哈希，哈希环多大，数据只存在一个节点，早晚有被打垮的时候。

1. map的实现原理

底层实现：红黑树

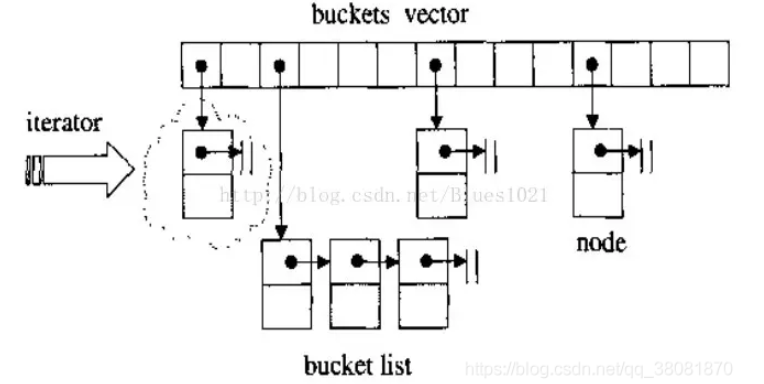
map的特性：map的所有元素都是pair，同时拥有key和value。所有元素都会根据元素的键值自动被排序。map不允许两个元素拥有相同的key值。pair是个结构体。

不可以通过map的迭代器改变map中的key值，因为map元素的key关系到map元素的排列规则。任意改变map的元素键值会严重破坏map组织。修改map的value值是可以的。

map、list对元素进行新增操作和删除操作，操作之前和之后的的所有迭代器都依然有效。当然，被删除的那个元素的迭代器除外。

map查找时间复杂为O（logN）

1. 源码剖析unordered\_map解决hash冲突的 过程（与哈希表的扩容机制、hash冲突时不同解决方法的优劣势、如何保证对unordered\_map线程安全的访问）



hash冲突：

1.线性探测 （慢性死亡，只是解决当下情况，后面会产生蝴蝶效应）

2.再哈希法 （使用哈希函数去散列一个输入，输出是同一个位置，再次哈希，直至不发生冲突）缺点：都要重新计算哈希，计算时间增加

3.开链法（优势：不会再有哈希冲突，内存消耗空间大，时间复杂度不会超过装载因子）

unordered\_map C++11正式加入的对hashmap的官方实现。

底层使用hashtable+buket（哈希表+桶）的实现原理，hashtable可以看作是一个数组或者vector之类的连续内存存储结构（可以通过下标来快速定位时间复杂度为O（1））处理

hash冲突的方法就是在相同的hash值得元素位置下面挂buket（桶），在使用链表来实现桶，即开链法实现hash冲突解决

所以查询一个树最差的时间复杂度是：首先进行一次hash运算找到桶的位置，然后使用链表来继续查找（所有元素在同一个桶里，其他桶位全为空，这个桶位其实就是一个数组下面挂红黑树也就是挂了一个map的结构）。所以时间复杂度是计算hash+O(1)+O(n)。但是这几乎是不可能的。在一个设计正常的hash函数里结果应该是偏向平均的，至少设计方向是偏向平均的。所以无论是查找效率还是插入、删除效率unordered\_map都优于map。所以在对数据不要求有序的情况下，尽量使用unordered\_map。除非你对数据要求有序才去使用map。

Hashtable的槽数一般都是根据需要分配的，且一般都是2的n次方大小。这种设计在计算桶号的优势就是可以通过使用按为&来加快运算。

int index = hash & （length - 1）

unordered\_map的插入过程：

1、得到 key；

2、通过 hash 函数得到 key对应的 hash 值；

3、得到桶号（一般都为 hash 值对桶数求模）；

4、存放 key 和 value 在桶内（发生冲突，用比较函数解决）。

unordered\_map的取值过程：

1、得到 key

2、通过 hash 函数得到 key对应的 hash 值

3、得到桶号（一般都为 hash 值对桶数求模）

4、比较桶内部是否有与 key 相等的元素，若都不相等，则没有找到；若存在，则取出相等的记录的。

如何保证对unordered\_map线程安全的访问？（提示：queue线程安全需要mutex）

读写锁（多线程读取的时候合理、写的时候仅允许一个线程进行写操作）

1. C++ STL map有几种，用什么实现的

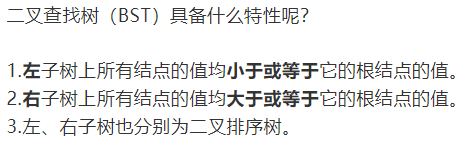
4种：map（红黑树）、multimap（红黑树中的insert\_equal而非insert\_unique）、unordered\_map(hashtable+buket开链法)、unordered\_multimap

1. 红黑树特点，于平衡二叉树的区别，跟一般搜索树的区别（要在RB\_Tree上及产能和红非叶子节点的删除操作，左旋右旋，旋转次数比较

<https://blog.csdn.net/z702143700/article/details/49079107>）

红黑树引用场景：多路复用技术的Epoll的核心结构也是红黑树+双向链表。Java中HashMap的底层实现。C++ map的底层实现

要学习红黑树，咱们需要先理解二叉查找树



二叉查找树的最坏情况，直接变成线性复杂度，查找能力大打折扣几乎变成线性。

如何解决二叉查找树多次插入新结点而导致的不平衡呢？我们的主角 [ 红黑树 ]应运而生。

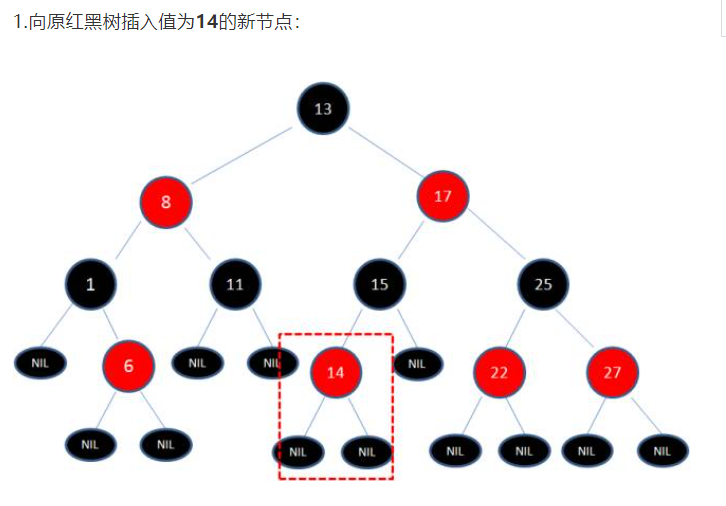
红黑树（Red Black Tree）是一种自平衡的二叉查找树。除了符合二叉查找树的基本特性外，它还具有下列附加特性：

1. 节点是红色或黑色
2. 根节点是黑色
3. 每个叶子节点都是黑色的空节点（NIL节点）
4. 每个红色节点的两个子节点都是黑色。（从每个叶子到根的所有路径不能有两个连续红色节点）
5. 从任一节点到每个叶子的所有路径都包含相同数目的黑色节点

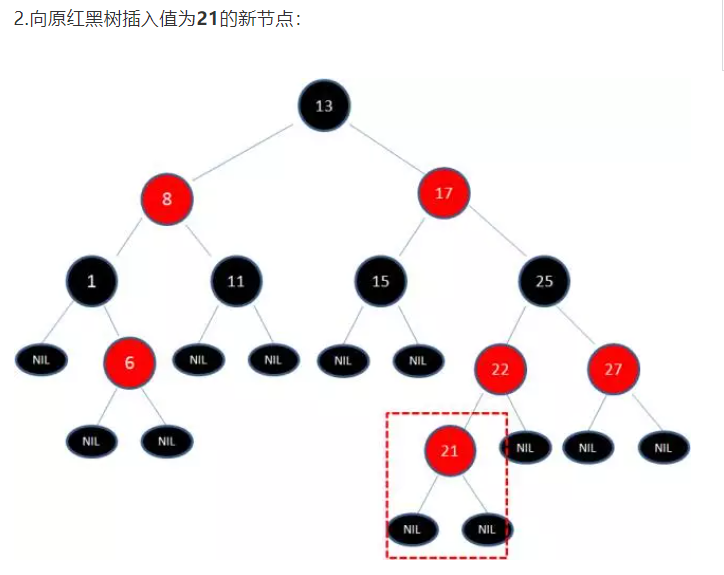




什么情况下会破坏红黑树的规则，什么情况下不会破坏规则呢？我们举两个简单的栗子：



由于父节点15是黑色节点，因此这种情况并不会破坏红黑树的规则，无需做任何调整。



由于父节点22是红色节点，因此这种情况打破了红黑树的规则4（每个红色节点的两个子节点都是黑色），必须进行调整，使之重新符合红黑树的规则。（变色和旋转）



1. 红黑树的结构以及如何调整树结构的
2. vector和set的区别

vector是顺序型容器，set是关联型容易一个key值只允许出现一次。

set的底层实现是红黑树，multiset底层也是红黑树（只不过中间一个方法调用不同。）

1. Vector底层实现，扩容，删除，push\_back和emplace\_back

vector 是最常用的容器之一，其底层所采用的数据结构非常简单，就只是一段连续的线性内存空间。



vector扩容的本质：

另外需要指明的是，当 vector 的大小和容量相等（size==capacity）也就是满载时，如果再向其添加元素，那么 vector 就需要扩容。vector 容器扩容的过程需要经历以下 3 步：

1. 完全弃用现有的内存空间，重新申请更大的内存空间；
2. 将旧内存空间中的数据，按原有顺序移动到新的内存空间中；
3. 最后将旧的内存空间释放。

这也就解释了，为什么 vector 容器在进行扩容后，与其相关的指针、引用以及迭代器可能会失效的原因。

由此可见，**vector 扩容是非常耗时的**。为了降低再次分配内存空间时的成本，每次扩容时 vector 都会申请比用户需求量更多的内存空间（这也就是 vector 容量的由来，即 capacity>=size），以便后期使用。

vector 容器扩容时，不同的编译器申请更多内存空间的量是不同的。以 VS 为例，它会扩容现有容器容量的 50%。在linux中，扩充是之前的2倍。

push\_back和emplace\_back

使用push\_back()向容器中加入一个右值元素（临时对象）的时候，首先会调用**构造函数**构造这个临时对象，然后需要调用**拷贝构造函数**将这个临时对象放入容器中。原来的临时变量释放。这样造成的问题是临时变量申请的资源就浪费。

使用emplace\_back，在容器尾部添加一个元素，这个元素原地构造，不需要触发拷贝构造和转移构造。

push\_back()右值时就会调用构造函数和转移构造函数

emplace\_back()函数向容器中加入临时对象， 临时对象原地构造，没有赋值或移动的操作。

emplace\_back()函数要比push\_back()函数要快一倍

1. gdb调试打断点命令是什么？（多线程gdb断点调试）






9. STL sort()底层实现