# 知识文档

1. 数据结构和算法
2. Linux基础
3. Java基础
4. 集合
   1. ArrayList

数组集合，默认10个长度，当容量达到最大值，按照1.5倍方式扩容。要求连续内存。随机访问get和set有优势。预留的尾部会浪费空间。

* 1. LinkedList

列表集合，不要求连续内存。新增add和删除remove有优势，因为arraylist需要移动数组数据。每个元素要保留下个节点信息浪费空间。

* 1. Vector

数组集合，默认10个长度，当容量达到最大值，按照2倍方式扩容。要求连续内存。随机访问get和set有优势。预留的尾部会浪费空间。所有方法synchronized同步关键字。

* 1. SynchronizedList

适用于多写少读。提供的list包装。

* 1. CopyOnWriteArrayList

每次写都复制一份新的数组，读没有锁，适用于多读少写。

* 1. HashSet

内部基于hashmap实现，屏蔽掉了value。

* 1. TreeSet

内部基于treemap实现，屏蔽掉了value。

* 1. LinkedHashSet

内部基于linkedHashmap实现，屏蔽掉了value。

* 1. ConcurrentHashSet

内部基于concurrentSkipListMap实现，屏蔽掉了value。

* 1. CopyOnWriteArraySet

内部基于copyOnWriteArrayList实现。

* 1. ConcurrentSkipListSet

内部基于concurrentSkipListMap实现，屏蔽掉了value。

* 1. EnumSet

相对枚举使用操作性能好。

* 1. HashMap

初始容量16，加载因子0.75(数值越大对空间的利用越好，但是查找效率下降。越小空间利用差，查找块)。数组+列表实现。允许key==null,value==null值保存在数组第一个位置。底层数组长度总是2的n次方（为了保证后续扩容）。Java8相对老版本扩容时,节点hash&N-1会有一定几率重新均匀打散节点存储，用hash高16位异或低16保证当数据长度较小时，也能保证高低位采用到hash运算，从而保证桶的利用率。当列表长度大于8时采用红黑树优化。

* 1. HashTable

初始容量11，加载因子0.75，扩容时n\*2+1保证基数，保证桶的利用率。方法上加synchronized，不允许key和value为null。

* 1. ConcurrentHashMap

初始容量16，加载因子0.75(数值越大对空间的利用越好，但是查找效率下降。越小，空间利用差，查找块。)。java8之前采用分段缩小锁的粒度提升性能和reentrantlock控制并发。Java8锁的粒度为node,和synchronized(node)控制并发。当列表数量大于8时采用红黑树优化。

* 1. LinkedHashMap

继承hashMap,利用列表记录了插入顺序（Entry before after）。

* 1. TreeMap

基于红黑树实现。不允许key为null。Key必需实现compare接口。

* 1. Properties

继承hashTable。

* 1. WeakHashMap

初始容量16，加载因子0.75(数值越大对空间的利用越好，但是查找效率 下降。越小，空间利用差，查找块。)。允许key为null,当某个key不在引用时会自动删除，当key的值被垃圾回收时也会被删除。节约空间，用来缓存非必需存在的数据。通过referenceQueue实现自动回收。每次操作时会先同步map中被回收的对象。2倍扩容。

* 1. ConcurrentSkipListMap

不允许值为null，支持二分查找的线程安全map。

* 1. EnumMap

提供枚举使用的一种快速map。因为容量大小固定为枚举的数量。

* 1. ArrayBlockingQueue

基于数组实现有界阻塞队列。初始化时必须提供一个固定容量，内部使用rentrantlock进行加锁。当调用off(add)时容量大于固定容量时，不插入返回false，否则插入返回true。调用put时，容量大于固定容量时，阻塞等待直到可以插入，put跟take对应。

* 1. ConcurrentLinkedQueue

基于列表实现无界非阻塞队列。基于cas解决并发。迭代弱一致。

* 1. LinkedBlockingQueue

基于列表实现无界队列，其他见ArrayBlockingQueue。

* 1. SynchronousQueue

无缓冲队列，必须生产者和消费者握手后才能一起离开。

* 1. Deque

双端队列，可以用来作栈先进后出，其他跟queue差不多，不作赘述。

1. 多线程相关知识
   1. Runnable接口
   2. Thread
   3. 线程池
      1. 单个线程池Executors.newSingleThreadExecutor，coreSize=1，maxSize=1，keepAlivetime=0。内部使用linkedBlockingQueue
      2. 固定大小线程池Executors.newFixedThreadPoll，coreSize=n，maxSize=n,keepAliveTime=0。内部使用linkedBlockingQueue
      3. 无限线程池Executors.newCachedThreadPoll,coreSize=0，maxSize=int.max，keepAliveTime=60秒。适用于任务周期短期，并发场景。内部使用SynchronousQueue。
      4. 调度线程池 Executors.newScheduledTheadPoll,coreSize=n，maxSize= int.max，keepAliveTime=0。内部使用delayedWorkQueue。
      5. ForkJionPoll Executors.newWorkStealingPoll，使用cpu可用线程数。内部forkJionPoll。
   4. 线程锁
      1. 并发安全特性
         1. 原子性

一个操作或多个操作要么全部执行完成且执行过程不被中断，要么就不执行。

* + - 1. 可见性

当多个线程同时访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

* + - 1. 有序性

程序执行的顺序按照代码的先后顺序执行。

* + 1. synchronized

jvm支持，jvm后期优化，修饰方法和方法块。隐式锁。

* + 1. volatile

修饰成员变量，借助cpu lock指令，保证内存可见性，现代多核cpu支持内存一致性协议，禁止指令重排。不适用于i++,因为i++不是原子指令。

* + 1. reentrantLock

可重入锁，基于aqs实现，支持公平和非公平。

* + - 1. 获取策略

Fair（公平锁），当锁的状态为0检查锁是否有下一个竞争者。

Nonfair(非公平),当锁的状态为0，直接参与cas修改状态竞争。

* + - 1. 获取锁
      2. 释放锁
    1. reentrantReadWriteLock

把读写可重入锁看成内部维护了2个reentrantLock锁，读锁共享一个公用的Thread实例作为锁的拥有线程，写锁使用当前线程作为锁的拥有线程。其他细节同参考reentrantLock。

* + 1. condition

配合reentrantlock使用，await当前线程等待和signal唤醒目标线程。

* + 1. stampedlock

java8新出，性能好于reentrantlock，但是不支持可重入，支持读写锁和乐观锁。

* + 1. countDownLatch

协调线程同步用，主线等待子线程全部完成，不可复用。

* + 1. cyclicbarrier

协调线程同步用，子线程等待所有子线程全部完成，可复用。

* + 1. 锁的优化

缩小锁的范围

缩小锁的粒度

通过业务手段，比如乐观锁，读写锁，分段锁提高并发性能。

1. 内存零拷贝知识
   1. FileChannnel

文件管道，在复制文件时，避免内核buffer复制，性能提升高。支持随机读写，和lock。

* 1. MappedByteBuffer

文件内存映射，读写性能极高，支持随机读写。打开的文件只有在gc时才会关闭，不确定性，需要借助反射方法关闭。

1. Io
   1. BIO

同步阻塞io。当前线程阻塞等待io完成。典型的一应一答线程比1:1，吞吐量低。编程模型难度低。适用于链接少，固定架构。

* 1. NIO

同步非阻塞io。采用多路复用selector模型。Selector监听多个文件具柄是否完成，轮询监听依旧是同步的，适用于链接多，链接时间短，比如聊天服务器。

* 1. AIO

异步非阻塞io。发起io操作后不需要等待,内核回调通知完成。链接多，且链接时间长（重操作），比如相册服务器，文件服务器。

* 1. socket

链接3次握手,向服务端发送syn j，服务端收到回复ack syn j+1，服务端向客户端发送syn k 客户端收到回复ack syn k+1。关闭4次挥手，向服务端发送fin m，服务端收到回复ack fin m+1,服务端通知上层应用，然后向客户端发送fin n，客户端收到回复ack服务端fin n+1。

1. jvm
   1. 内存分布
      1. 堆（方法区）

所有线程共享的区域，存放对象的，eden(年轻代),survivor（from,to），old。当堆无法扩展时抛outOfMemoryError异常。

* + 1. 线程栈

线程私有，生命期同线程一样，不需要回收，先进后出。当无法申请到栈的时抛outOfMemoryError异常，当请求栈的深度大于所允许的深度时抛stackOverFlowErroe。

* + 1. 本地方法栈

java调用本地方法时的栈。

* + 1. 程序计数器

线程私有，记录当前线程所执行的指令指示器。

* 1. 如何判断对象是否可回收算法
     1. 对象引用计数，实现简单，但是无法解决循环引用
     2. 判断对象gc root是否可达
        1. 虚拟机栈中引用的对象
        2. 方法区中静态类属性引用的对象
        3. 方法区中常量引用的对象
        4. 本地方法栈中jni引用的对象
  2. JVM对象回收算法
     1. 标记清理：快，简单，产生垃圾碎片。
     2. 复制清理：快，简单。不产生垃圾碎片，但是可用空间少。
     3. 分代标记整理清理：复杂。
  3. JVM垃圾回收器
     1. 串行回收器
     2. 并行回收器
     3. CMS回收器
     4. G1回收器

1. java代码优化方式
   1. 尽量指定类、方法的final修饰符

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用final的例子，例如java.lang.String，整个类都是final的。为类指定final修饰符可以让类不可以被继承，为方法指定final修饰符可以让方法不可以被重写。如果指定了一个类为final，则该类所有的方法都是final的。Java编译器会寻找机会内联所有的final方法，内联对于提升Java运行效率作用重大，具体参见Java运行期优化。此举能够使性能平均提高50%。

* 1. 尽量重用对象

特别是String对象的使用，出现字符串连接时应该使用StringBuilder/StringBuffer代替。由于Java虚拟机不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理，因此，生成过多的对象将会给程序的性能带来很大的影响。

* 1. 尽可能使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中速度较快，其他变量，如静态变量、实例变量等，都在堆中创建，速度较慢。另外，栈中创建的变量，随着方法的运行结束，这些内容就没了，不需要额外的垃圾回收。

* 1. 及时关闭流

Java编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销，稍有不慎，将会导致严重的后果。

* 1. 及时关闭流

Java编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销，稍有不慎，将会导致严重的后果。

* 1. 尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

for (int i = 0; i < list.size(); i++)

{...}

建议替换为：

for (int i = 0, int length = list.size(); i < length; i++)

{...}

这样，在list.size()很大的时候，就减少了很多的消耗

* 1. 尽量采用懒加载的策略，即在需要的时候才创建

比如通过内部类实现单利，利用classload实现了懒加载

* 1. 慎用异常

异常对性能不利。抛出异常首先要创建一个新的对象，Throwable接口的构造函数调用名为fillInStackTrace()的本地同步方法，fillInStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java虚拟机就必须调整调用堆栈，因为在处理过程中创建了一个新的对象。异常只能用于错误处理，不应该用来控制程序流程。

* 1. 不要在循环中使用try…catch…，应该把其放在最外层

除非不得已。如果毫无理由地这么写了，只要你的领导资深一点、有强迫症一点，八成就要骂你为什么写出这种垃圾代码来了。

* 1. 如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度
  2. 当复制大量数据时，使用System.arraycopy()命令
  3. 乘法和除法使用移位操作
  4. 循环内不要不断创建对象引用
  5. 基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList
  6. 尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销
  7. 不要将数组声明为public static final
  8. 尽量在合适的场合使用单例
  9. 尽量避免随意使用静态变量（要知道，当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的）
  10. 及时清除不再需要的会话
  11. 实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历
  12. 使用同步代码块替代同步方法（这点在多线程模块中的synchronized锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率）
  13. 将常量声明为static final，并以大写命名（这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量）
  14. 不要创建一些不使用的对象，不要导入一些不使用的类
  15. 程序运行过程中避免使用反射

关于，请参见反射。反射是Java提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制，特别是Method的invoke方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存—-用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

* 1. 使用数据库连接池和线程池
  2. 使用带缓冲的输入输出流进行IO操作(带缓冲的输入输出流，即BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升IO效率)
  3. 顺序插入和随机访问比较多的场景使用ArrayList，元素删除和中间插入比较多的场景使用LinkedList
  4. 不要让public方法中有太多的形参
  5. 字符串变量和字符串常量equals的时候将字符串常量写在前面
  6. 请知道，在java中if (i == 1)和if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问，”if (i == 1)”和”if (1== i)”有没有区别，这就要从C/C++讲起。

在C/C++中，”if (i == 1)”判断条件成立，是以0与非0为基准的，0表示false，非0表示true，如果有这么一段代码：

int i = 2;

if (i == 1)

{

...

}else{

...

}

C/C++判断”i==1″不成立，所以以0表示，即false。但是如果：

int i = 2;if (i = 1) { ... }else{ ... }

万一程序员一个不小心，把”if (i == 1)”写成”if (i = 1)”，这样就有问题了。在if之内将i赋值为1，if判断里面的内容非0，返回的就是true了，但是明明i为2，比较的值是1，应该返回的false。这种情况在C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在if语句中不正确的赋值操作，建议将if语句写为：

int i = 2;if (1 == i) { ... }else{ ... }

这样，即使开发者不小心写成了”1 = i”，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值i为1，但是不能对一个常量赋值1为i。

但是，在Java中，C/C++这种”if (i = 1)”的语法是不可能出现的，因为一旦写了这种语法，Java就会编译报错”Type mismatch: cannot convert from int to boolean”。但是，尽管Java的”if (i == 1)”和”if (1 == i)”在语义上没有任何区别，但是从阅读习惯上讲，建议使用前者会更好些。

* 1. 不要对数组使用toString()方法
  2. 不要对超出范围的基本数据类型做向下强制转型
  3. 公用的集合类中不使用的数据一定要及时remove掉
  4. 把一个基本数据类型转为字符串，基本数据类型.toString()是最快的方式、String.valueOf(数据)次之、数据+””最慢
  5. 对资源的close()建议分开操作

1. 缓存知识
2. 缓存穿透

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

* 1. 将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力
  2. 另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

1. 缓存雪崩

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

* 1. 缓存失效时的雪崩效应对底层系统的冲击非常可怕。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。这里分享一个简单方案就时讲缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

1. 缓存击穿

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

3.1 业界比较常用的做法，是使用分布式锁。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

1. 开源框架的学习
2. netty
3. RPC
4. redis
5. zookeeper
6. kafka
7. elasticsearch
8. 分布式知识
9. paxos算法
10. 选举算法
11. 可用性
12. 一致性
13. 网络分区
14. 负载均衡
    1. routing table(路由表或者映射表)
       1. 优劣势
15. 简单，全局维护一张table即可。
16. 单点故障。
17. 因为table记录了每个节点的信息，所以可以达到节点均衡。
18. 性能较差，首先需要查询路由然后再访问目标节点。
19. 节点扩展方便。
    * 1. 优化方式
20. stand-by避免单点故障。
21. 路由表索引优化，提高查询性能
22. 客户端初始化时，保存一份路由表快照。采取主动或者被动方式更新，当客户端本地路由表查询路由信息时再到服务端查询路由新，减少服务端路由查询。
    1. 哈希算法(murmurhash3和horner法则)
       1. 优劣势
          1. 性能好，在客户段根据某个字段直接计算出目标节点。
          2. 节点扩展差，需要重新分配数据。
          3. 存在单节点过热。以redis3.2.8为例，生产环节就出现过单节点过热。影响整个服务性能。
       2. 优化方式
          1. 引入虚拟节点环，可以加强节点离线或者上线容错
          2. 物理节点对应的虚拟节点分布均匀
          3. 虚拟节点环每次增加n个物理节点时，重新调整虚拟节点环数。保证最终虚拟节点数为2的n次方。这个可以参考java8 hashmap扩容算法
       3. 算法评估标准
          1. 数据分布

一个衡量的措施是考虑一个哈希函数是否能将一组数据的哈希值进行很好的分布。要进行这种分析，需要知道碰撞的哈希值的个数，如果用链表来处理碰撞，则可以分析链表的平均长度，也可以分析散列值的分组数目。

* + - 1. 哈希函数的效率

另个一个衡量的标准是哈希函数得到哈希值的效率。通常，包含哈希函数的算法的算法复杂度都假设为O(1)，这就是为什么在哈希表中搜索数据的时间复杂度会被认为是"平均为O(1)的复杂度"，而在另外一些常用的数据结构，比如图(通常被实现为红黑树)，则被认为是O(logn)的复杂度。一个好的哈希函数必修在理论上非常的快、稳定并且是可确定的。通常哈希函数不可能达到O(1)的复杂度，但是哈希函数在字符串哈希的线性的搜索中确实是非常快的，并且通常哈希函数的对象是较小的主键标识符，这样整个过程应该是非常快的，并且在某种程度上是稳定的。

* + - 1. 平衡性(Balance)

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

* + - 1. 单调性(Monotonicity)

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

* + - 1. 分散性(Spread)

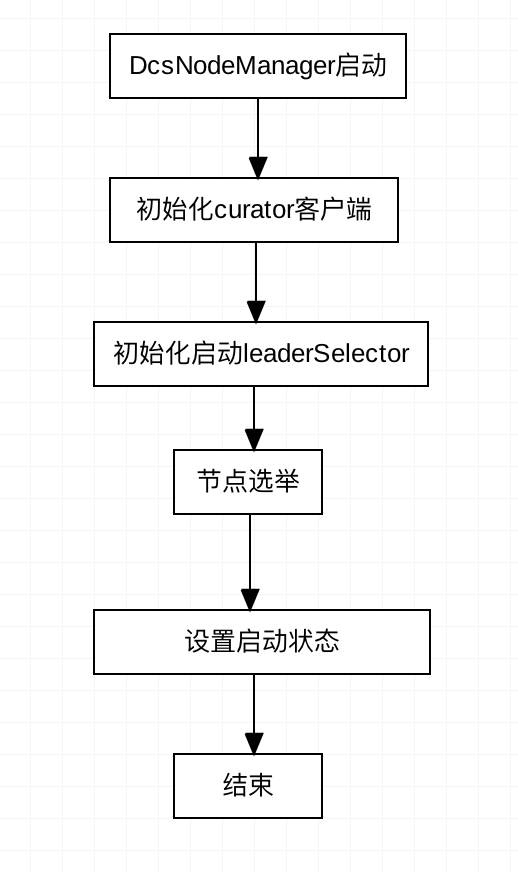
在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

* + - 1. 负载(Load)

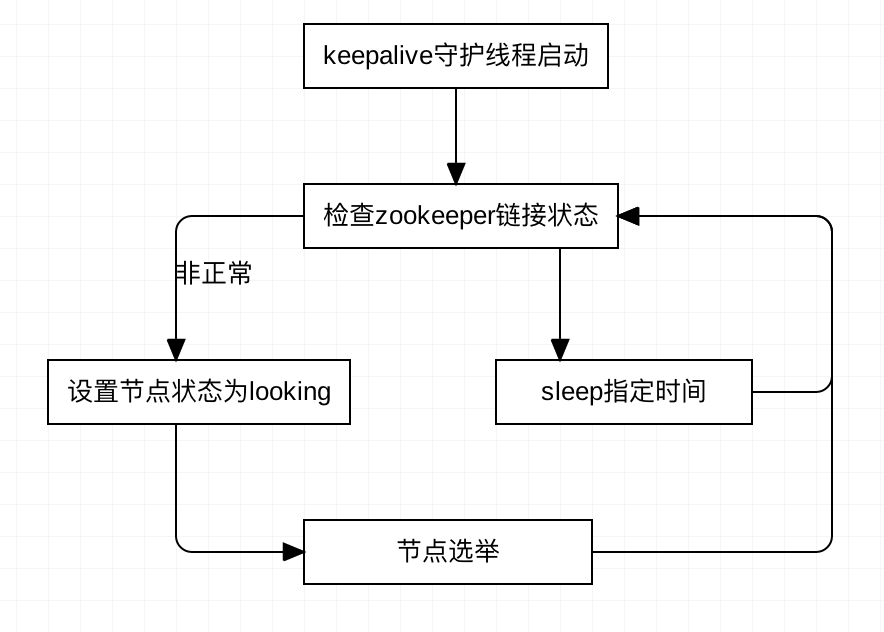
负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

在这篇文章中介绍的哈希函数被称为简单的哈希函数。它们通常用于散列（哈希字符串）数据。它们被用来产生一种在诸如哈希表的关联容器使用的key。这些哈希函数不是密码安全的，很容易通过颠倒和组合不同数据的方式产生完全相同的哈希值。

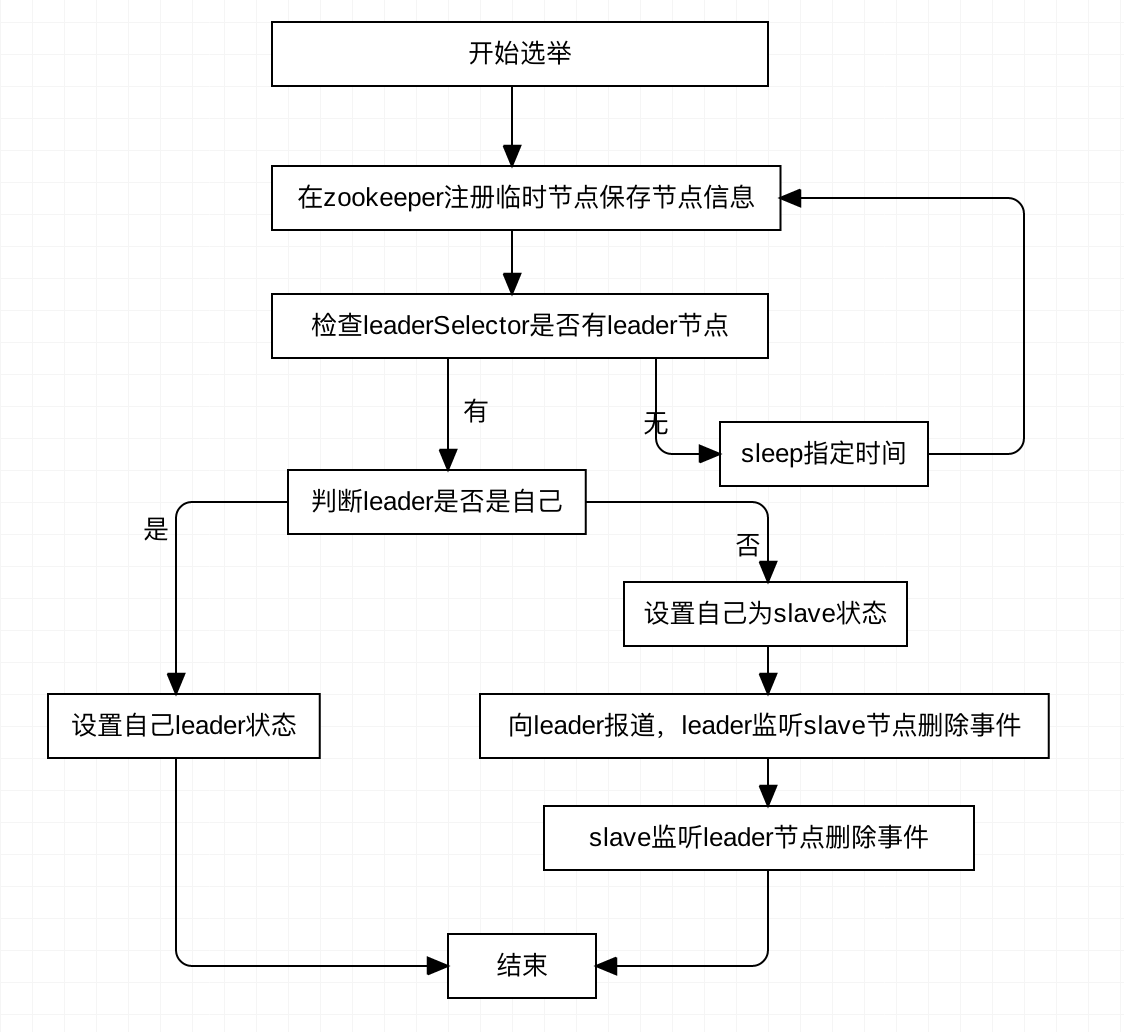
1. 节点发现机制
2. 项目经验
3. 分布式节点管理
   1. 目标
      1. 支持节点动态横向扩展。
      2. 提供分布式主从模型。
      3. 提供分布式锁。
      4. 自动维护主节点选举和当主节点挂掉重新选举。
      5. 提供节点事件watcher机制，比如成为主节点，加入从节点，丢失主节点，丢失从节点等事件。(watcher机制可以很友好的与上层应用解耦)。
   2. 设计
      1. 启动流程图



* + 1. keepalive流程图



* + 1. 选举流程图



* 1. 容错机制
     1. 节点状态looking,leader,slave，保证节点处于leader和slave状态下才能提供正常服务，looking状态下暂停服务。
     2. 基于zookeeper watcher节点机制监听离线节点。
     3. leader节点宕机，集群其他节点自动触发选举，直到从新选举出leader节点，理论上每个节点都有可能成为leader。

1. 分布式任务调度
   1. 目标
      1. 支持cron定时，手动，弹性时间（滑动窗口），任务关系，其他自定义条件触发。
      2. 支持FIFO,公平，优先级抢占,优先级非抢占，配额等多种调度策略。
      3. 最大化集群资源利用率。
      4. 任务执行等待最小（低延时执行）。
      5. 集群任务分布负载均衡。
      6. 保证集群稳定。
      7. 具备良好的节点可扩展性。
      8. 具备良好的容错机制。
   2. 设计
   3. 资源调度器
      1. 功能
         1. 基于分布式节点管理实现。
         2. 支负责集群系统资源的收集和管理。
         3. 维护集群系统状态。
         4. 对上层应用提供系统状态事件watch机制。（比如当系统资源紧张状态时，注册停止一些非紧急任务，保障系统稳定性。）
   4. 任务调度器
      1. 任务类型
         1. 内存作业
         2. 离线作业
         3. 流失作业
         4. 迭代式作业
         5. 爬虫作业
         6. web server作业
      2. 调度策略
         1. FIFO策略
            1. 优势

实现简单

先进先出原则

* + - * 1. 劣势

不够灵活

业务单一（体现不出分组和优先级概念）

不能充分利用集群资源

* + - 1. 公平策略
         1. 优势

实现简单

每个人任务都能得到一样的资源

* + - * 1. 劣势

不够灵活

业务单一（体现不出分组和优先级概念）

每次执行任务需要重新公平分配资源

* + - 1. 优先级抢占和非抢占策略
         1. 优势

灵活，支持多种任务

相同级任务FIFO

支持抢占低级任务资源

* + - * 1. 劣势

实现复杂

* + - 1. 配额分组策略
         1. 优势

灵活，支持各种任务分组

设置min值（0-min值资源昂贵）和max值（min-max共享资源相对廉价）

执行任务时，在有资源的条件下，资源不能超过max

执行任务时，必须保障min资源可用

当其他任务执行时，min-max段资源可被共享

* + - * 1. 劣势

实现复杂

* + - 1. 组合策略
         1. 优势

以上多种策略组合

* + - * 1. 劣势

实现复杂

* + 1. 技术要点
       1. 数据Locality
       2. 数据Shuffle
       3. Instance重跑和backup instance
  1. 任务触发器
     1. 手动触发
     2. 定时触发
        1. 支持cron定时表达式
     3. 任务关系触发
        1. 支持a任务执行时触发b任务执行
        2. 支持a任务执行完触发b任务执行
     4. 自定义条件触发
     5. 弹性时间触发
        1. 支持滑动窗口机制
  2. 任务执行器
  3. 容错机制
     1. 规模：两层架构易于横向扩展，资源管理和调度模块仅负责资源的整体分配，不负责具体任务调度，可以轻松扩展集群节点规模。
     2. 容错：当某个任务运行失败不会影响其他任务的执行；同时资源调度失败也不影响任务调度
     3. 扩展性：不同的计算任务可以采用不同的参数配置和调度策略，同时支持资源抢占
     4. 调度效率：计算framework决定资源的生命周期，可以复用资源，提高资源交互效率
     5. leader通知slave执行任务容错
        1. 重试通知slave执行任务。
        2. 多次失败后通知其他slave执行，当某个slave上

instance运行周期长，那么可以采用backup- instance,谁先执行完那么采用谁的结果。（当前instance运行时间远远超过其他instance时间。

数据处理速度低于其他instance平均值。完成的 instance比例。）

* + 1. slave节点挂掉容错
       1. leader重试通知slave执行任务。
       2. 通知其他节点执行执行该任务。
       3. slave节点挂掉后，leader将挂掉节点上的任务转移至其他slave执行。
    2. leader节点挂掉容错
       1. leader节点状态check point，snapshot(运行快照)定期持久化。
       2. slave向新的leader重新汇报
       3. leader节点stand-by
       4. 硬状态持久化。重新拉起时通过持久化恢复
       5. 软状态要求其他节点发送一份挂点前的信息
  1. 规模挑战
     1. 横向扩展和收缩
        1. 节点动态发现机制
        2. 节点动态离线机制
     2. 增量的消息通讯，减少通讯链路
        1. 独立通信甬道
        2. 增量的资源调度
     3. 异步持续多线程
        1. 执行线程的复用。
     4. 基于规则和历史信息的错误节点检测机制，降低这些错误对应用的影响
  2. 安全和性能隔离
     1. kvm
     2. docker
     3. lvs

1. 分布式爬虫
   1. 目标
      1. 简单页面智能化
      2. 通用页面配置化
      3. 复杂页面插件化
   2. 防爬策略
      1. ip访问频率

通过ip池和控制好每个ip访问随机频率

* + 1. 用户账号访问频率和用户常登录地址

通过多用户和匹配的ip

* + 1. 数据js加载

根据数据业务重要性和数据量去分析js或者chrome headless模式解决

* + 1. js加密跳转

根据数据业务重要性和数据量去分析js或者chrome headless模式解决

* + 1. 验证码

简单的字符识别或者1加2等于几验证码直接写识别算法处理，复杂的比如滑动或者点汉子通过购买第三方打码平台处理

* + 1. 数据图片化

ocr识别技术。

* + 1. 请求token机制

根据数据业务重要性和数据量去分析js或者chrome headless模式解决

* + 1. 人为行为识别

模拟人为鼠标键盘浏览习惯等行为

* + 1. Robots协议
       1. URL过滤器
       2. URL地图
       3. 网站访问频率
  1. 设计
     1. 流程图
  2. 容错机制
     1. 对每次任务下载的页面进行指定时间的保存
        1. 解决http304状态文档无更新，取缓存页面
        2. 为后续爬到的数据跟页面数据不一致做参照
     2. 支持http request重定向，最多不能超过指定max，否则抛出相关异常
     3. 202或者403http状态码切换ip
     4. 当爬虫行为被目标网站识别并返回验证码页面时（针对不需要登登录的网站）提供ip切换机制
     5. 当出现502，503或者504http状态码时，进行重试机制，当大量出现502，503或者504http状态码时暂停当前网站采集，让出资源供其他任务执行，并做check poion或者snapshot以备后续恢复
     6. 维护多个抽取规则，并进行排名。跟排名等级迭代进行抽取，直到抽取到内容，否则抛出相关异常
     7. 对页面抽取出的信息提供检验机制，保证数据的准确性和完整性
     8. 统计采集任务访问站点的时间和异常，为后续任务弹性调度做参考基础
     9. 统计每次采集任务执行的运行记录，异常信息和统计信息作为任务执行记录持久化，为后续任务监控和问题排查做依据

#### 页面自动识别

* + 1. 基于人眼对于屏幕的视线（相对居中），排除最左最右两端，取x坐标URL最多的作为内容页URL 。oschina和csdn博客测试通过。
    2. 版块URL的识别
    3. 基于对各种类型网站版块库进行识别。
    4. 新闻内容或者其他文章识别
       1. 标题识别
       2. 发表时间识别
       3. 正文识别
          1. 基于符合人的视野位置提取
          2. 正文包含主谓宾等

1. 架构设计
2. 设计模式
3. 异常定义和处理
4. 事件机制
5. 事务日志机制
6. 容错总结