jdk是java规范的实现,sun jdk是目前使用最广泛的jdk。

jvm原理、调优

池技术：对象池、线程池、连接池

反射：性能不好，替代技术，字节码技术

nio；

多线程：同步、异步

集合对象：各种集合对象的实现原理，特点

越基础的东西越重要

书籍：think in java

框架：shiro、spring、springmvc、spring-security、spring-session、mybatis、hibernate

编程基础：数据结构、算法

# JDK常用的几个包

## 集合包

集合包中包含了Collection和Map两个接口的实现类。

### ArrayList

默认构造器为ArrayList（）；该构造器会调用ArrayList（10）；来对集合进行初始化，10就是初始容量。

#### 使用ArrayList的注意要点

* ArrayList是基于数组方式实现的，无容量的限制。
* ArrayList在执行插入元素时可能要扩容，在删除元素时并不会减少数组的容量，如果希望响应的缩小数组的容量，可以调用ArrayList的trimToSize（）方法，在查找元素时要遍历数组，对于非null的元素采取equals的方式寻找。
* ArrayList是非线程安全的。

#### ArrayList中的一些坑

* ArrayList.toArray方法的返回值中的每一个元素都是转换成了Object类型,不能直接将返回的结果强制转换成其他类型的数组.如果想实现toArray之后得到一个目标类型的数组,可以使用toArray(T[] a)方法,该方法会将list中的元素存放到传入的a变量中,当然有一点需要注意,传入的a对象的长度必须大于等于list的长度,否则还是返回Object数组.
* 使用Arrays获取到的List是一个Arrays的内部类ArrayList对象,该ArrayList继承了AbstractList,但只重写了部分方法,如果调用该ArrayList的add方法会执行AbstractList的add方法,该方法直接抛出一个异常

### LinkedList

JDK7中LinkedList不是闭环存储了，添加了first和last变量，分别指向第一个元素和最后一个元素

### Vector

Vector和ArrayList基本相同,不同点就是Vector的方法上基本都有synchronized关键字,是线程安全的,并且可以指定扩容策略.

### Stack

Stack继承自Vector,不同的是Stack实现了LIFO(先进先出)的功能.

# Exception

## Exception对象

Exception类继承自Throwable类,在Throwable的构造方法中保存了一份堆栈信息,但只保存了方法名和类名,不能获取堆栈中方法的参数值

## UncaughtExceptionHandler

UncaughtExceptionHandler是java提供的一种错误捕捉机制,给我们的程序绑定一个UncaughtExceptionHandler实例之后,如果程序中抛出了异常,在程序中没有进行捕捉,那么会执行UncaughtException实例中的uncaughtException方法

UncaughtExceptionHandler是一个接口,使用时定义自己的实现类,然后调用Thread类的静态方法setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler)即可.

# Thread

## shutdownHook

shutdownHook是java提供的一种虚拟机关闭时触发一段逻辑的机制。可以调用runtime的addShutdownHook方法指定虚拟机关闭时要执行的代码，该方法需要一个Thread实例作为参数

## volatile

valatile是保证成员变量在多线程情况下保证变量可见性的一个关键字

场景:

如果线程1引用了成员变量,然后在线程2中对该成员变量做了修改,那么线程1是感知不到该成员变量的修改的.如果使用volatile修饰该成员变量,那么可以保证线程1对该变量值的变化的可见性.

# ThreadPool

## Executors

Java通过Executors提供四种线程池，分别为：

* newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
* newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
* newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
* newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

newCachedThreadPool和newFixedThreadPool方法返回的都是ThreadPoolExecutor实例.

newSingleThreadExecutor方法返回的是FinalizableDelegatedExecutorService实例,

newScheduledThreadPool方法返回的是ScheduledThreadPoolExecutor实例

## ThreadPoolExecutor

ThreadPoolExecutor是ExecutorService的默认实现

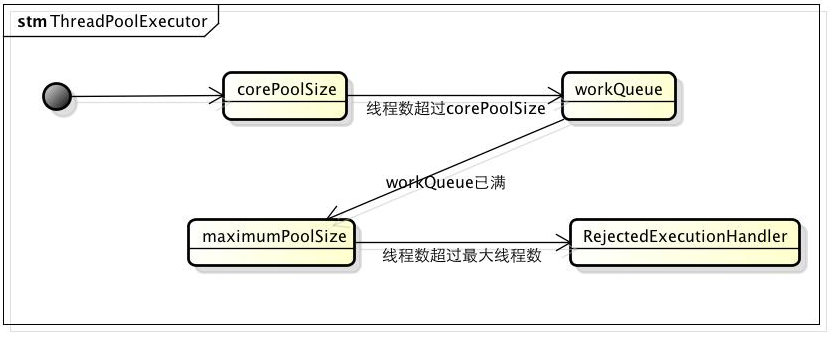
### 构造方法

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,long keepAliveTime, TimeUnit unit,BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler)

* corePoolSize： 线程池维护线程的最少数量
* maximumPoolSize：线程池维护线程的最大数量
* keepAliveTime： 线程池维护线程所允许的空闲时间
* unit： 线程池维护线程所允许的空闲时间的单位
* workQueue： 线程池所使用的缓冲队列
* handler： 线程池对拒绝任务的处理策略

### corePoolSize，maximumPoolSize，workQueue之间关系

* 当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。
* 当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行
* 当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务
* 当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理
* 当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程
* 当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭



## 大并发任务量时ThreadPool可能出现的问题

### newCachedThreadPool

newCachedThreadPool方法返回的是一个corePoolSize为0,maximumPoolSize为Integer.MAX\_VALUE, BlockingQueue为SynchronousQueue(无容量队列)的ThreadPoolExecutor对象,所以当任务来临时会不停的创建新的线程,并发任务量过大时可能就造成系统崩溃了.

### newFixedThreadPool

newFixedThreadPool方法返回的是一个corePoolSize和maximumPoolSize为指定的值,BlockingQueue为LinkedBlockingQueue(容量无边界) 的ThreadPoolExecutor对象,所以当任务来临时,如果正在使用的线程数达到了corePoolSize,那么会将任务压入LinkedBlockingQueue中,并发量过大时可能造成内存溢出.

所以在并发任务量很大的场景中可以使用ThreadPoolExecutor自定义自己的线程池,指定一个有边界的队列,并指定RejectedExecutionHandler,给出当任务量超过线程池的处理能力时的处理策略

# Servlet3.0新特性

1. 异步处理支持

通过异步处理特性,Servlet线程不再需要一直阻塞(直到业务处理完毕才能再输出响应,最后才结束该Servlet线程).在接收到请求之后,Servlet线程可以将耗时的操作委派给另一个线程来完成,自己在不生成响应的情况下返回至容器

1. 新增的注解支持

新增了若干注解,用于简化Servlet、Filter、Listener的生命，这使得web.xml部署描述文件从该版本开始不再是必须的了.

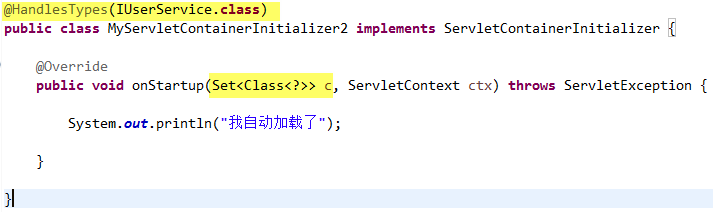
1. 可插性支持

通过该特性的支持,开发者可以将自己的功能打成jar包放到工程的lib文件夹下,容器将自动加载其中的组件

可插性支持的实现是通过引入"Web模块部署描述符片段"的web-fragment.xml部署描述文件.该文件必须存放在JAR文件的META-INF目录下,该部署描述文件可以包含一切可以在web.xml中定义的内容.

# ServletContainerInitializer的使用

ServletContainerInitializer是Servlet3.0中新添加的一种初始化容器的一种机制,在容器启动时,容器会通过java SPI机制加载ServletContainerInitializer的实现类,并执行onStartUp方法.如下图:



onStartup方法会获取两个参数,Set<Class<?>>和ServletContext对象,其中Set<Class<?>>对象可以通过@HandlesTypes注解指明希望接收的类,如果不指定,Set<Class<?>>的值为null

Spring利用这种特性,提供了自己的实现SpringServletContainerInitializer, 这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成,AbstractAnnotationConfigDispatcherServletInitializer类实现了WebApplicationInitializer.所以我们继承AbstractAnnotationConfigDispatcherServletInitializer并重写其中的方法,即可实现Spring的初始化,这样就不需要在web.xml中配置DispatcherServlet和ContextLoaderListener了.

利用这种特性,我们可以提供自己的Initializer,处理自己的逻辑,比如开启webservice服务等

# java SPI机制

SPI机制是一种服务发现机制

为了实现在模块装配的时候能在程序里动态指明,这就需要一种服务发现机制.java spi就是提供这样的一个机制:为某个接口寻找服务实现的机制.有点类似IOC的思想,就是将装配的控制权移到程序之外,在模块化设计中这个机制尤其重要.

## java SPI的具体约定

当服务的提供者提供了服务接口的一种实现之后,在jar包的META-INF/services/目录里同时创建一个以服务接口命名的文件.该文件里就是实现该服务接口的具体实现类.而当外部程序装配这个模块的时候,就能通过该jar包META-INF/services/里的配置文件找到具体的实现类名,并装载实例化,完成模块的注入.

jdk提供的服务查找的一个工具类:java.util.ServiceLoader

# jmx

JMX的全称为Java Management Extensions.是管理java的一种扩展.这种机制可以方便的管理正在运行的java程序.常用于管理线程、内存、日志Level、服务重启、系统环境等。

一个MBean是一个被管理的java对象，一个设备、一个应用或者任何资源都可以被表示为MBean，MBean会暴露一个接口对外,这个接口可以读取或者写入一些对象中的属性,通常一个MBean需要定义一个接口,以MBean结尾.如EchoMBean

# java nio

Java NIO 由以下几个核心部分组成：

* Channels
* Buffers
* Selectors

虽然Java NIO 中除此之外还有很多类和组件，但在我看来，Channel，Buffer 和 Selector 构成了核心的API。其它组件，如Pipe和FileLock，只不过是与三个核心组件共同使用的工具类。因此，在概述中我将集中在这三个组件上。其它组件会在单独的章节中讲到。

## Channel 和 Buffer

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。这里有个图示：



Channel和Buffer有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

正如你所看到的，这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。

与这些类一起的有一些有趣的接口，但为简单起见，我尽量在概述中不提到它们。本教程其它章节与它们相关的地方我会进行解释。

以下是Java NIO里关键的Buffer实现：

* ByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

这些Buffer覆盖了你能通过IO发送的基本数据类型：byte, short, int, long, float, double 和 char。

Java NIO 还有个 MappedByteBuffer，用于表示内存映射文件， 我也不打算在概述中说明。

## Selector

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：



要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

# java buffer

## Buffer类基本概念

　 　一般而言，Buffer的数据结构是一个保存了原始数据的数组，在Java语言里面封装成为一 个带引用的对象。Buffer一般称为缓冲区，该缓冲区的优点在于它虽然是一个简单数组，但是它封装了很多数据常量以及单个对象的相关属性。针对 Buffer而言主要有四个主要的属性：

* 容 量（Capacity ）： 容量描述了这个缓冲区最 多能够存放多少，也是Buffer的最大存储元素量，这个值是在创建Buffer的时候指定的，而且不可以更改
* 限 制（Limit ）： 不能够进行读写的缓冲区 的第一个元素，换句话说就是这个Buffer里面的活动元素数量
* 位 置（Position ）： 下一个需要进行读写的元 素的索引，当Buffer缓冲区调用相对get()和set()方法的时候会自动更新Position的值
* 标记（ Mark ）： 一个可记忆的 Position位置的值，当调用mark()方法的时候会执行mark = position，一旦调用reset()的时候就执行position = mark,和Position有点不一样，除非进行设置，否则Mark值是不存在的。

　　按照上边的对应关系可以知道：

0 <= mark <= position <= limit <= capacity

### Buffer的基本操 作：

　　Buffer管理（Accessing）：

　　一般情况下Buffer可以管理很多元素，但是在程序开发过程中我们 只需要关注里面的活跃 元素 ， 如上图小于limit位置的这些元素，因为这些元素是真正在IO读写过程需要的。当Buffer类调用了put()方法的时候，就在 原来的Buffer中插入了某个元素，而调用了get()方法过后就调用该位置的活跃元素，取出来进行读取，而Buffer的get和put方法一直很神 秘，因为它存在一个相对和绝对的概念：

　　在相对版本 的put 和get 中，Buffer本身不使用index 作为参数，当相对方法调用的时候，直接使用position作为基点，然后运算 调用结果返回，在相对操作的时候，如果position的值过大就有可能抛出异常 信息；同样的相对版本的put方法调用的时候当调用元素超越了limit 的限制的时候也会抛出BufferOverflowException 的 异常 ，一般情况为：position > limit 。

　　在绝对版本 的put和 get 中，Buffer的position却不会收到影响，直接使用index 进行调用，如果index越界的时候直接抛出 Java里面常见的越界异常 ：java.lang. IndexOutOfBoundException 。

　　针对get和put方法的两种版本的理解可以查阅API看看方法 get的定义【这里查 看的是Buffer类的子类ByteBuffer的API】 ：

public abstract byte get() throws BufferUnderflowException

public ByteBuffer get(byte [] dst) throws BufferUnderflowException

public ByteBuffer get(byte [] dst,int offset,int length) throws BufferUnderflowException,IndexOutOfBoundException

public abstract byte get(int index) throws IndexOutOfBoundException

　　从上边的API详解里面可以知道，Buffer本身支持的两种方式的访问是有原因 的，因为Buffer本身的设计目的是为了使得数据能够更加高效地传输，同样能够在某一个时刻移动某些数据。当使用一个数组作为参数的时候，整个 Buffer里面的 position位置放置了一个记录用的游标，该游标不断地在上一次操作完结的基础 上 进行移动来完成Buffer本身的数据的读取，这种情况下一般需要提 供一个length参数，使用该参数的目的就是为了防止越界操作的发生。如果请求的数据没有办法进行传输，当读取的时候没有任何数据能够读取的时候，这个 缓冲区状态就不能更改了，同时这个时候就会抛出BufferUnderflowException的异常 ， 所以在向缓冲区请求的时候使用数组结构存储时， 如果没有指定length参数，系统会默认为填充整个数组的长度，这种情况和上边IO部分的缓冲区的设置方法类似。也就是说当编程过程需要将一个 Buffer数据拷贝到某个数组的时候（这里可以指代字节数组），需要显示指定拷贝的长度，否则该数组会填充到满，而且一旦当满足异常 条件：即limit 和position不匹配的时候，就会抛异常 。

# 一致性hash

## 介绍

一致性hash算法在1997年有麻省理工学院提出,设计目标是为了解决因特网中的热点问题.一致性hash解决了动态的网络拓扑中分布式存储和路由.

**优点**

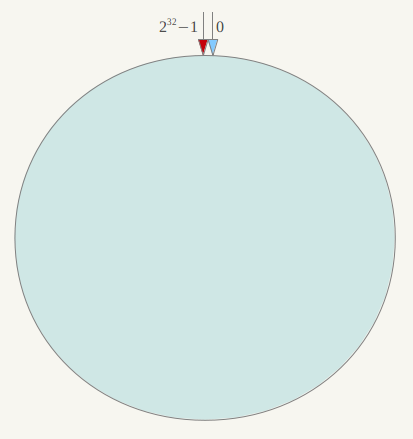
当系统中添加节点后只需要移动很少的内容就可以打到平衡

**不足**

一致性hash不足的地方就是,在查询过程中,需要经过O(N)步才能到达被查询的节点,因为要一一和节点的hash进行比对,当系统规模非常大时,节点数量可能超过百万,这样的查询效率显然难以满足使用的需求.

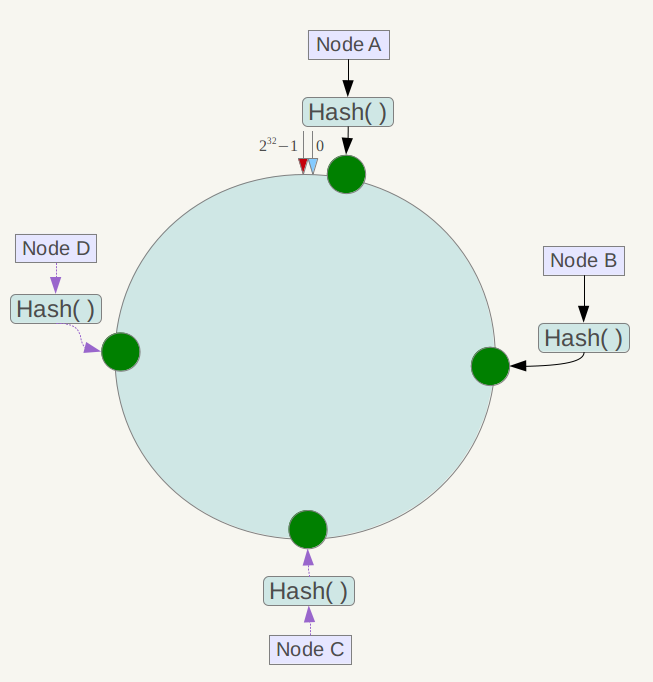
## 原理

一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^32-1（即哈希值是一个32位无符号整形），整个哈希空间环如下：



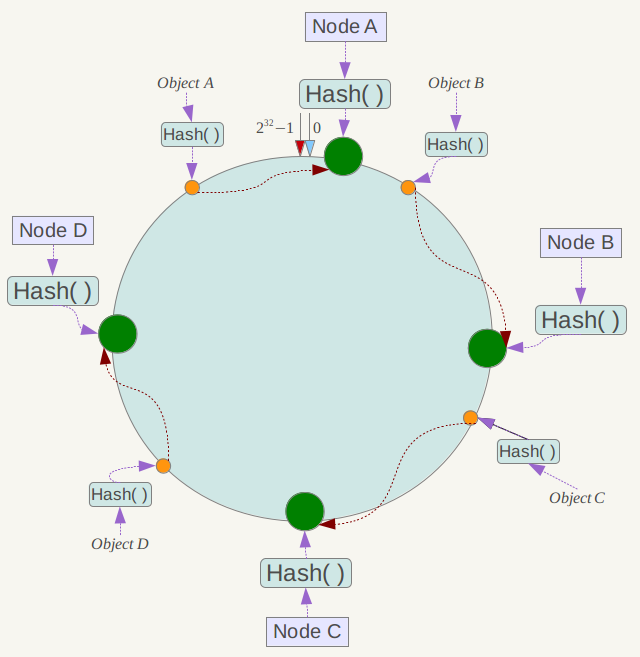
整个空间按顺时针方向组织。0和232-1在零点中方向重合。

　　下一步将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用ip地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。

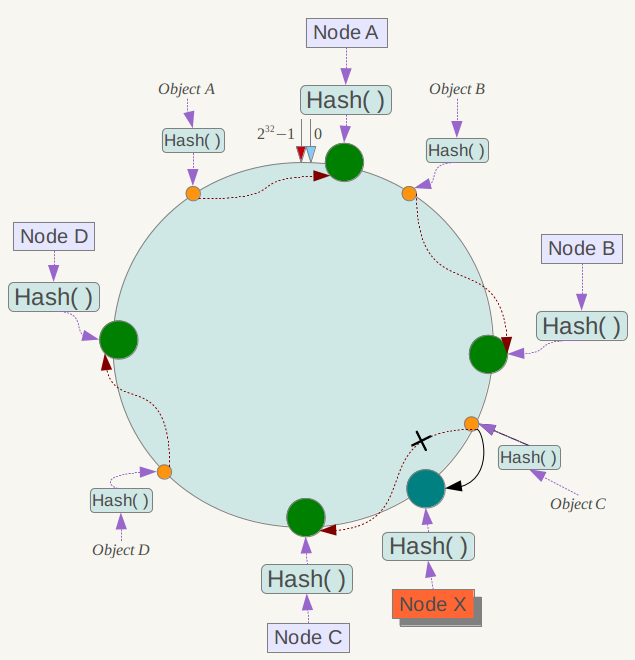
　　例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

下面分析一致性哈希算法的容错性和可扩展性。现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

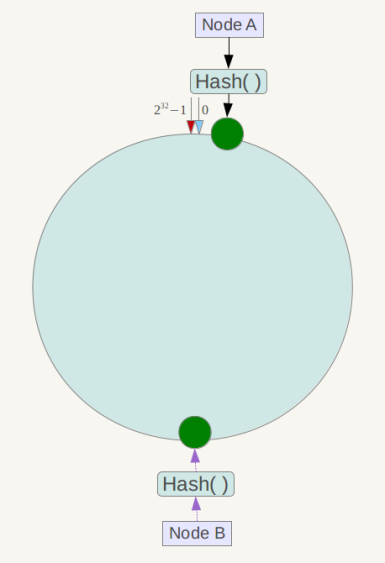
下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



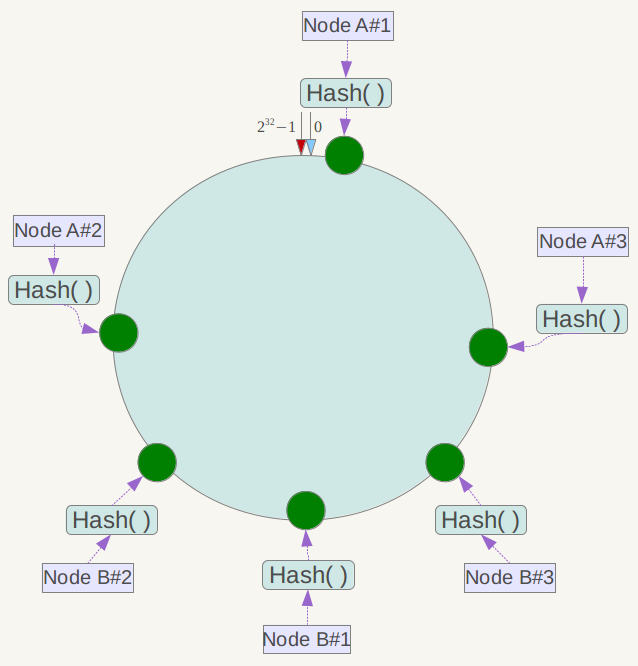
此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

另外，一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。例如系统中只有两台服务器，其环分布如下，



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，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，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。