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，如果希望相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.

# 类加载

## 类字节码Class文件

Class文件由类装载器装载后，在JVM中将形成一份描述Class结构的元信息对象，通过该元信息对象可以获知Class的结构信息：如构造函数，属性和方法等，Java允许用户借由这个Class相关的元信息对象间接调用Class对象的功能。

## 类加载机制

虚拟机把描述类的数据从class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

## 类加载器

类装载器就是寻找类的字节码文件，并构造出类在JVM内部表示的对象组件。

(1) Bootstrap ClassLoader : 将存放于<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar 名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用

     (2) Extension ClassLoader : 将<JAVA\_HOME>\lib\ext目录下的，或者被java.ext.dirs系统变量所指定的路径中的所有类库加载。开发者可以直接使用扩展类加载器。

     (3) Application ClassLoader : 负责加载用户类路径(ClassPath)上所指定的类库,开发者可直接使用。

## 类加载过程

在Java中，类装载器把一个类装入JVM中，要经过以下步骤：

(1) 装载：查找和导入Class文件；

(2) 链接：把类的二进制数据合并到JRE中；

    a)校验：检查载入Class文件数据的正确性；

    b)准备：给类的静态变量分配存储空间；

    c)解析：将符号引用转成直接引用；

(3) 初始化：对类的静态变量，静态代码块执行初始化操作

### 类初始化

      (1) 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候，读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

      (2) 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

      (3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

      (4)当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

只有上述四种情况会触发初始化，也称为对一个类进行主动引用，除此以外，所有其他方式都不会触发初始化，称为被动引用

代码清单1

|  |
| --- |
| **publicclass** SupperClass {  **static**{  System.***out***.println("I am supper class");  }  **publicstatic** String *words* = "--SupperClass--";  }  **publicclass** SubClass **extends** SupperClass{  **static**{  System.***out***.println("I am SubClass");  }  }  **publicclass** TestBody {  **publicstaticvoid** main(String[] args) {  System.***out***.println(SubClass.*words*);  }  } |

上述代码运行后，只会输出【I am supper class】, 而不会输出【I am SubClass】,对于静态字段，只有直接定义这个字段的类才会被初始化,因此，通过子类来调用父类的静态字段，只会触发父类的初始化,但是这是要看不同的虚拟机的不同实现

代码清单2

|  |
| --- |
| **publicclass** ConstClass {  **static**{  System.***out***.println("I am ConstClass");  }  **publicstaticfinal** String ***value*** = "const1";  }  **publicclass** MainTest {  **publicstaticvoid**main(String[] args) {  System.***out***.println(ConstClass.***value***);  }  } |

对常量ConstClass.value 的引用实际都被转化为NotInitialization类对自身常量池的引用，这两个类被编译成class后不存在任何联系。所以运行MainTest后只会输出”const1”而不会输出"I am ConstClass".

### 装载

    在装载阶段，虚拟机需要完成以下3件事情

        (1) 通过一个类的全限定名来获取定义此类的二进制字节流

        (2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构

        (3) 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

    虚拟机规范中并没有准确说明二进制字节流应该从哪里获取以及怎样获取,这里可以通过定义自己的类加载器去控制字节流的获取方式。

### 验证

    虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统奔溃。

### 准备

    准备阶段是正式为类变量分配并设置类变量初始值的阶段，这些内存都将在方法区中进行分配,需要说明的是：

这时候进行内存分配的仅包括类变量(被static修饰的变量),而不包括实例变量,实例变量将会在对象实例化时随着对象一起分配在Java堆中;这里所说的初始值“通常情况”是数据类型的零值，假如:

public static int value = 123;

value在准备阶段过后的初始值为0而不是123,而把value赋值的putstatic指令将在初始化阶段才会被执行

## 反射

Reflection机制允许程序在正在执行的过程中，利用Reflection APIs取得任何已知名称的类的内部信息,包括:

package、 type parameters、 superclass、 implemented interfaces、 inner classes、 outer classes、 fields、 constructors、 methods、 modifiers等，

并可以在执行的过程中，动态生成instances、变更fields内容或唤起methods。

# 常用的类

## Class

一个Class对象代表一个字节码对象,在java中如果向创建一个对象,需要先加载该类的字节码文件即Class对象,然后再将该类实例化为对象

### isAssignableFrom

A. isAssignableFrom(B)用于判断A是否是B的子类或子接口

## ClassLoader

用于加载字节码文件(.class文件)

## Date

|  |  |  |
| --- | --- | --- |
|  | 日期 | 时间 |
| java.util.Date | 有 | 有 |
| java.sql.Date | 有 | 无 |
| java.sql.Timestamp | 有 | 有 |
| java.sql.Time | 无 | 有 |

# 注解

## @Inherited

允许子类继承父类的注解

### 测试

添加@Inherited注解的注解

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***)  @Target(ElementType.***TYPE***)  @Inherited  **public@interface**InheritedAnnotation {  **public** String name() **default**"1";  } |

未添加@Inherited注解的注解

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***)  @Target(ElementType.***TYPE***)  **public@interface**NoInheritedAnnotation {    **public** String name() **default**"2";  } |

父类

|  |
| --- |
| @InheritedAnnotation  @NoInheritedAnnotation  **publicclass** Super {  } |

子类

|  |
| --- |
| **publicclass** Sub **extends** Super {  } |

测试类

|  |
| --- |
| @Test  **publicvoid** testInheritedAnnotation(){  //获取子类字节码  Class<Sub>clazz = Sub.**class**;  //获取子类上所有注解  Annotation[] annotations = clazz.getAnnotations();  **for** (Annotation annotation : annotations) {  Class<? **extends** Annotation>annotationType = annotation.annotationType();  System.***out***.println(annotationType.getName());  }  //输出结果为: com.annotation.InheritedAnnotation  }    @Test  **publicvoid** testInheritedAnnotation2(){  //获取父类字节码  Class<Super>clazz = Super.**class**;  //获取父类上所有注解  Annotation[] annotations = clazz.getAnnotations();  **for** (Annotation annotation : annotations) {  Class<? **extends** Annotation>annotationType = annotation.annotationType();  System.***out***.println(annotationType.getName());  }  //输出结果为: com.annotation.InheritedAnnotation  com.annotation.NoInheritedAnnotation  } |

# 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.

## heap dump

-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath=/path/file.hprof 来指定文件的输出路径

## 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

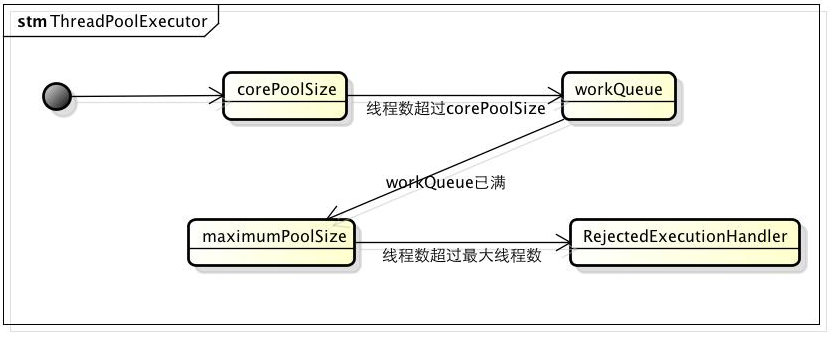
#### 构造方法

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

# Java EE

## 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.

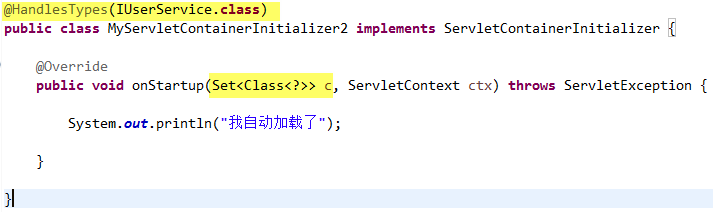
1. 可插性支持

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

在socket，nio中的有些API中，申请的内存是直接向OS要的，在堆中分析内存是查看不到的，可以通过-XX:MaxDirectMemorySize=<size>来设置应用向OS直接申请的最大内存数。

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**

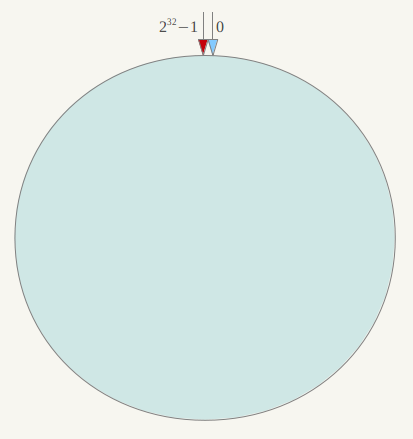
当系统中添加节点后只需要移动很少的内容就可以打到平衡

**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.

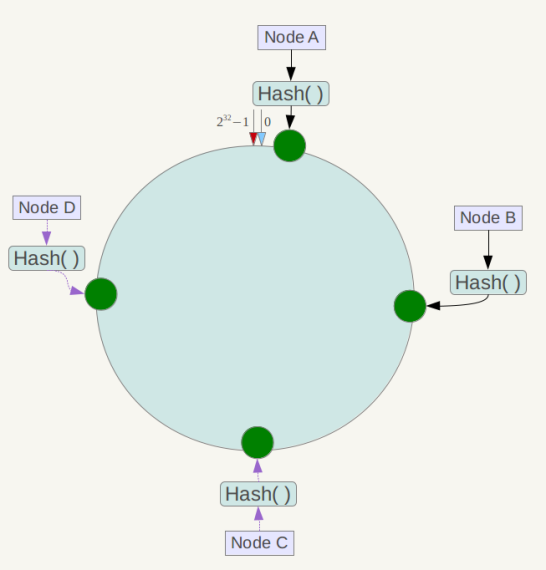
## 原理

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：



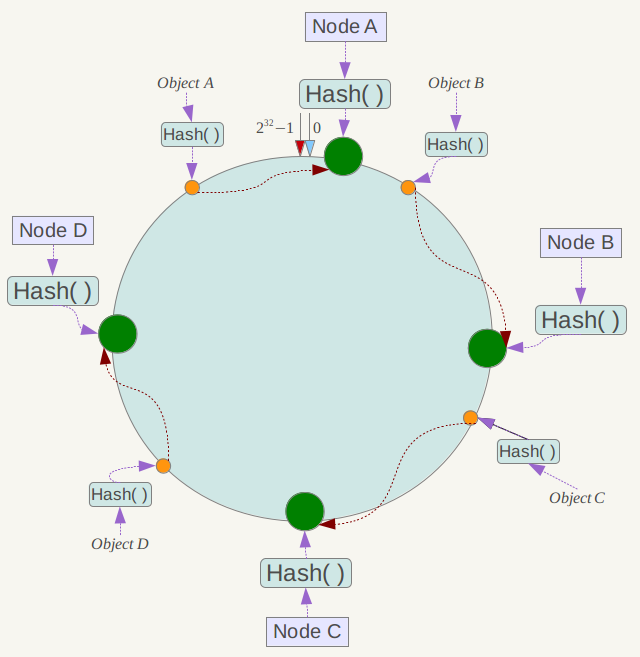
整个空间按顺时针方向组织。0和232-1在零点中方向重合。

　　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。

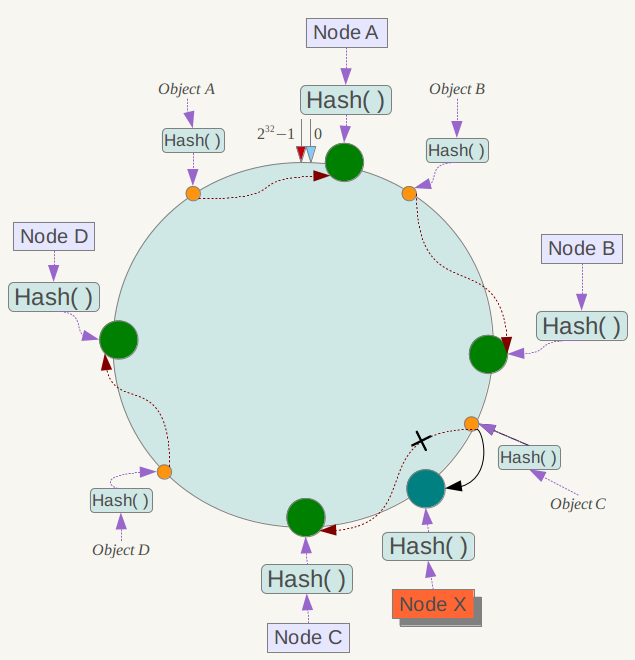
　　例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



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。

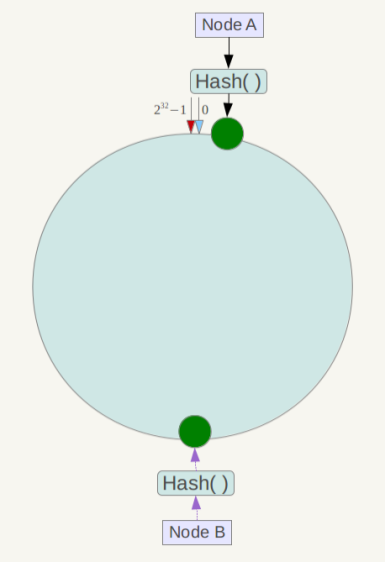
下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



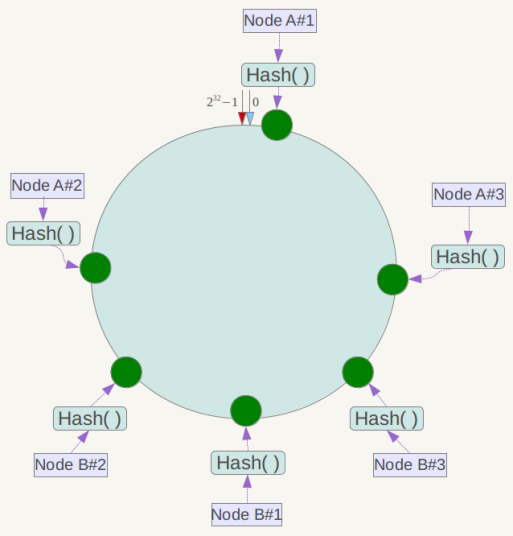
此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

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。

# jvm

## java和javaw命令

both applications will run your program, and are very similar, but have one key difference - java runs it through a Java Console, while javaw does not. The console is there mostly for text programs - for example, all your print statememe...

javaw不会在控制台打印任何信息

如果程序是纯粹的图形化的，而且你需要提升速度，用javaw。

## jvm常用参数

-D开头的参数用于设置系统变量

-Xnoclassgc

关闭虚拟机对 class 的垃圾回收功能。

-Xloggc:<file>

将虚拟机每次垃圾回收的信息写到日志文件中，文件名由 file 指定，文件格式是平文 件，内容和-verbose:gc 输出内容相同。

-Xms<size>

设置虚拟机可用内存堆的初始大小，缺省单位为字节，该大小为 1024 的整数倍并且要 大于1MB，可用 k(K)或m(M)为单位来设置较大的内存数。初始堆大小为 2MB。

例如：-Xms6400K，-Xms256M

-Xmx<size>

设置虚拟机内存堆的最大可用大小，缺省单位为字节。该值必须为 1024 整数倍，并且 要大于 2MB。可用 k(K)或 m(M)为单位来设置较大的内存数。缺省堆最大值为 64MB。

例如：-Xmx81920K，-Xmx80M

-Xss<size>

设置线程栈的大小，缺省单位为字节。与-Xmx 类似，也可用 K 或 M 来设置较大的值。 通常操作系统分配给线程栈的缺省大小为 1MB

-verbose:gc

在虚拟机发生内存回收时在输出设备显示信息，格式如下：

[Full GC 268K->168K(1984K), 0.0187390 secs]

该参数用来监视虚拟机内存回收的情况。

-verbose:jni

在虚拟机调用 native 方法时输出设备显示信息，格式如下：

[Dynamic-linking native method HelloNative.sum ... JNI]

该参数用来监视虚拟机调用本地方法的情况，在发生 jni错误时可为诊断提供便利。

-client，-server

这两个参数用于设置虚拟机使用何种运行模式，client 模式启动比较快，但运行时性能 和内存管理效率不如 server 模式，通常用于客户端应用程序。相反，server 模式启动 比 client慢，但可获得更高的运行性能。

在 windows上，缺省的虚拟机类型为 client 模式，如果要使用 server模式，就需要在 启动虚拟机时加-server 参数，以获得更高性能，对服务器端应用，推荐采用 server 模 式，尤其是多个 CPU 的系统。在 Linux，Solaris 上缺省采用 server模式。

-classpath,-cp

虚拟机在运行一个类时，需要将其装入内存，虚拟机搜索类的方式和顺序如下：

Bootstrap classes，Extension classes，User classes。

Bootstrap 中的路径是虚拟机自带的 jar 或 zip 文件，虚拟机首先搜索这些包文件，用

System.getProperty("sun.boot.class.path")可得到虚拟机搜索的包名。

Extension 是位于 jre\lib\ext 目录下的 jar 文件，虚拟机在搜索完 Bootstrap 后就搜索 该目录下的 jar 文件。用 System. getProperty("java.ext.dirs”)可得到虚拟机使用 Extension的搜索路径。

User classes 搜索顺序为当前目录、环境变量 CLASSPATH、-classpath。

## 内存间交互操作

* lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
* unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
* read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
* load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
* use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
* assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
* store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
* write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

write

store

load

read

临时存储

变量

assign

use

变量副本

临时存储

主内存

工作内存

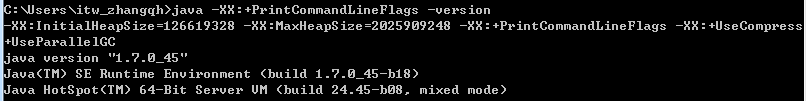
执行引擎

工作内存和主内存交互图

## 虚拟机参数查看

### -XX:+PrintCommandLineFlags参数

显示出JVM初始化完毕后所有跟最初的默认值不同的参数及它们的值。



### -XX:+PrintFlagsFinal参数

可以获取到所有可设置参数及值(手动设置之后的值)，这个参数只能使用在Jdk6 update 21以上版本(包括该版本)。

### 堆(Heap)和非堆(Non-heap)内存

按照官方的说法：“Java虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。可以看出JVM主要管理两种类型的内存：堆和非堆。简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

堆内存分配：

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；

JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4。

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；

空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。

因此服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小。

非堆内存分配：

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；

由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。

JVM内存限制(最大值)

首先JVM内存限制于实际的最大物理内存，假设物理内存无限大的话，JVM内存的最大值跟操作系统有很大的关系。简单的说就32位处理器虽然可控内存空间有4GB,但是具体的操作系统会给一个限制，这个限制一般是2GB-3GB（一般来说Windows系统下为1.5G-2G，Linux系统下为2G-3G），而64bit以上的处理器就不会有限制了。

设置VM参数导致程序无法启动主要有以下几种原因：

1) 参数中-Xms的值大于-Xmx，或者-XX:PermSize的值大于-XX:MaxPermSize；

2) -Xmx的值和-XX:MaxPermSize的总和超过了JVM内存的最大限制，比如当前操作系统最大内存限制，或者实际的物理内存等等。

说到实际物理内存这里需要说明一点的是，如果你的内存是1024MB，但实际系统中用到的并不可能是1024MB，因为有一部分被硬件占用了。

### java7中常量池的变更

从java7开始,常量池有perm space移到heap中

|  |
| --- |
| Area: HotSpot  Synopsis: In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the String.intern() method will see more significant differences. |

## heap dump

|  |
| --- |
| A heap dump is a snapshot of the memory of a Java process at a certain point of time. There are different formats for persisting this data, and depending on the format it may contain different pieces of information, but in general the snapshot contains information about the java objects and classes in the heap at the moment the snapshot was triggered. **Usually a full GC is triggered before the heap dump is written so it contains information about the remaining objects.** |

根据上面的描述可以看到在存储dump文件之前会触发一次full GC,所以被dump捕捉到的对象在做dump时都是不可被回收的对象

## java 引用类型

参考:http://zhangjunhd.blog.51cto.com/113473/53092/

在JDK 1.2以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。这4种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

### 强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

### 软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存（下文给出示例）。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

### 弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

### 虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

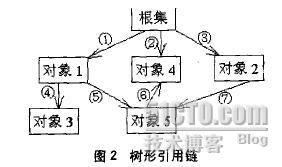
虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

### 为什么需要使用软引用

首先，我们看一个雇员信息查询系统的实例。我们将使用一个Java语言实现的雇员信息查询系统查询存储在磁盘文件或者数据库中的雇员人事档案信息。作为一个用户，我们完全有可能需要回头去查看几分钟甚至几秒钟前查看过的雇员档案信息(同样，我们在浏览WEB页面的时候也经常会使用“后退”按钮)。这时我们通常会有两种程序实现方式:一种是把过去查看过的雇员信息保存在内存中，每一个存储了雇员档案信息的Java对象的生命周期贯穿整个应用程序始终;另一种是当用户开始查看其他雇员的档案信息的时候，把存储了当前所查看的雇员档案信息的Java对象结束引用，使得垃圾收集线程可以回收其所占用的内存空间，当用户再次需要浏览该雇员的档案信息的时候，重新构建该雇员的信息。很显然，第一种实现方法将造成大量的内存浪费，而第二种实现的缺陷在于即使垃圾收集线程还没有进行垃圾收集，包含雇员档案信息的对象仍然完好地保存在内存中，应用程序也要重新构建一个对象。我们知道，访问磁盘文件、访问网络资源、查询数据库等操作都是影响应用程序执行性能的重要因素，如果能重新获取那些尚未被回收的Java对象的引用，必将减少不必要的访问，大大提高程序的运行速度。

## 对象可及性的判断

在很多时候，一个对象并不是从根集直接引用的，而是一个对象被其他对象引用，甚至同时被几个对象所引用，从而构成一个以根集为顶的树形结构。



在这个树形的引用链中，箭头的方向代表了引用的方向，所指向的对象是被引用对象。由图可以看出，从根集到一个对象可以由很多条路径。比如到达对象5的路径就有①-⑤，③-⑦两条路径。由此带来了一个问题，那就是某个对象的可及性如何判断:

* 单条引用路径可及性判断:在这条路径中，最弱的一个引用决定对象的可及性。
* 多条引用路径可及性判断:几条路径中，最强的一条的引用决定对象的可及性。

比如，我们假设图2中引用①和③为强引用，⑤为软引用，⑦为弱引用，对于对象5按照这两个判断原则，路径①-⑤取最弱的引用⑤，因此该路径对对象5的引用为软引用。同样，③-⑦为弱引用。在这两条路径之间取最强的引用，于是对象5是一个软可及对象。

# 常用概念

## 跨域资源共享

CORS

http://www.ruanyifeng.com/blog/2016/04/cors.html

* CORS是一个W3C标准,全称是"跨域资源共享"(Cross-origin resource sharing)
* 它允许浏览器向跨源服务器,发出XMLHttpRequest请求,从而克服AJAX只能同源使用的限制.
* CORS需要浏览器和服务器端都支持才能使用,需要IE10或更高版本
* CORS支持多种请求方式GET,PUT,DELETE等

jsonp

* jsonp也是用于实现跨域请求的
* jsonp对浏览器版本没有要求
* jsonp只支持GET请求

## 幂等操作

在编程中,一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同.幂等函数,或幂等方法,是指可以使用相同的参数重复执行,并能获得相同结果的函数.这些函数不会影响系统状态,也不用担心重复执行会对系统造成改变.

## 查看端口占用情况

windows

netstat -aon会列出所有端口占用情况

# 设计模式

## 状态设计模式

## 策略设计模式

为了找到if ..else if ..else语句的替代解决方案,找到了策略设计模式

参考文章: http://blog.csdn.net/hguisu/article/details/7558249/

### 场景

从生活中的场景来类比策略设计模式的使用场景

例如 出行旅游：我们可以有几个策略可以考虑：可以骑自行车，汽车，做火车，飞机。每个策略都可以得到相同的结果，但是它们使用了不同的资源。选择策略的依据是费用，时间，使用工具还有每种方式的方便程度 。

### 定义

策略模式：定义一系列的算法(即策略),把每一个算法封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。也称为政策模式(Policy)。

### 适用性

当存在以下情况时使用Strategy模式

1）• 许多相关的类仅仅是行为有异。 “策略”提供了一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。

2）• 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间 /时间权衡的算法。当这些变体实现为一个算法的类层次时 ,可以使用策略模式。

3）• 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

4）• 一个类定义了多种行为 , 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

### 策略模式优点

1) 相关算法系列 Strategy类层次为Context定义了一系列的可供重用的算法或行为。 继承有助于析取出这些算法中的公共功能。

2) 提供了可以替换继承关系的办法： 继承提供了另一种支持多种算法或行为的方法。你可以直接生成一个Context类的子类，从而给它以不同的行为。但这会将行为硬行编制到 Context中，而将算法的实现与Context的实现混合起来,从而使Context难以理解、难以维护和难以扩展，而且还不能动态地改变算法。最后你得到一堆相关的类 , 它们之间的唯一差别是它们所使用的算法或行为。 将算法封装在独立的Strategy类中使得你可以独立于其Context改变它，使它易于切换、易于理解、易于扩展。

3) 消除了一些if else条件语句 ：Strategy模式提供了用条件语句选择所需的行为以外的另一种选择。当不同的行为堆砌在一个类中时 ,很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的Strategy类中消除了这些条件语句。含有许多条件语句的代码通常意味着需要使用Strategy模式。

4) 实现的选择 Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间 /空间权衡取舍要求从不同策略中进行选择。

### 策略模式缺点

1)客户端必须知道所有的策略类，并自行决定使用哪一个策略类:  本模式有一个潜在的缺点，就是一个客户要选择一个合适的Strategy就必须知道这些Strategy到底有何不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时 , 才需要使用Strategy模式。

2 ) Strategy和Context之间的通信开销 ：无论各个ConcreteStrategy实现的算法是简单还是复杂, 它们都共享Strategy定义的接口。因此很可能某些 ConcreteStrategy不会都用到所有通过这个接口传递给它们的信息；简单的 ConcreteStrategy可能不使用其中的任何信息！这就意味着有时Context会创建和初始化一些永远不会用到的参数。如果存在这样问题 , 那么将需要在Strategy和Context之间更进行紧密的耦合。

3 )策略模式将造成产生很多策略类：可以通过使用享元模式在一定程度上减少对象的数量。 增加了对象的数目 Strategy增加了一个应用中的对象的数目。有时你可以将 Strategy实现为可供各Context共享的无状态的对象来减少这一开销。任何其余的状态都由 Context维护。Context在每一次对Strategy对象的请求中都将这个状态传递过去。共享的 Strategy不应在各次调用之间维护状态。

### 策略模式角色

这个模式涉及到三个角色：

* **环境(Context)角色：**持有一个Strategy的引用。
* **抽象策略(Strategy)角色：**这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
* **具体策略(ConcreteStrategy)角色：**包装了相关的算法或行为。

### 策略模式样例

环境角色

|  |
| --- |
| **public class** Context {  //持有一个具体策略的对象  **private** Strategy strategy;  /\*\*  \* 构造函数，传入一个具体策略对象  \* @param strategy 具体策略对象  \*/  **public** Context(Strategy strategy){  this.strategy = strategy;  }  /\*\*  \* 策略方法  \*/  **public void** contextInterface(){    strategy.strategyInterface();  }    } |

抽象策略类

|  |
| --- |
| **public interface** Strategy {  /\*\*  \* 策略方法  \*/  **public void** strategyInterface();  } |

具体策略类

|  |
| --- |
| **public class** ConcreteStrategyA implements Strategy {  **@Override**  **public void** strategyInterface() {  //相关的业务  }  } |

|  |
| --- |
| **public class** ConcreteStrategyB implements Strategy {  **@Override**  **public void** strategyInterface() {  //相关的业务  }  } |

|  |
| --- |
| **public class** ConcreteStrategyC implements Strategy {  **@Override**  **public void** strategyInterface() {  //相关的业务  }  } |

### 策略模式和状态模式

策略模式和其它许多设计模式比较起来是非常类似的。策略模式和状态模式最大的区别就是策略模式只是的条件选择只执行一次，而状态模式是随着实例参数（对象实例的状态）的改变不停地更改执行模式。换句话说，策略模式只是在对象初始化的时候更改执行模式，而状态模式是根据对象实例的周期时间而动态地改变对象实例的执行模式。

•可以通过环境类状态的个数来决定是使用策略模式还是状态模式。

•策略模式的环境类自己选择一个具体策略类，具体策略类无须关心环境类；而状态模式的环境类由于外在因素需要放进一个具体状态中，

以便通过其方法实现状态的切换，因此环境类和状态类之间存在一种双向的关联关系。

•使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环境类的状态会根据用户的操作自动转换。

•如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式；

如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式。