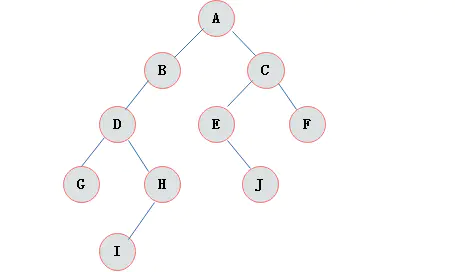
**Java知识点总结**

# Java基础

## 数据结构

### 二叉树

参考地址：https://www.jianshu.com/p/bf73c8d50dc2



#### 特点

1. 每个结点最多有两颗子树，所以二叉树中不存在度大于2的结点。
2. 左子树和右子树是有顺序的，次序不能任意颠倒。
3. 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树

#### 性质

1. 在二叉树的第i层上最多有2i-1 个节点 。（i>=1）
2. 二叉树中如果深度为k,那么最多有2k-1个节点。(k>=1）
3. n0=n2+1 n0表示度数为0的节点数，n2表示度数为2的节点数。
4. 在完全二叉树中，具有n个节点的完全二叉树的深度为[log2n]+1，其中[log2n]是向下取整。
5. 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点有如下特性：

(1) 若 i=1，则该结点是二叉树的根，无双亲, 否则，编号为 [i/2] 的结点为其双亲结点;

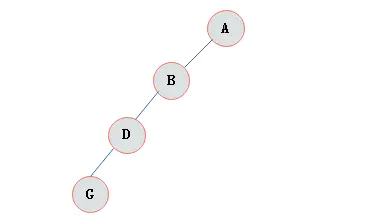
(2) 若 2i>n，则该结点无左孩子， 否则，编号为 2i 的结点为其左孩子结点；

(3) 若 2i+1>n，则该结点无右孩子结点， 否则，编号为2i+1 的结点为其右孩子结点。

#### 特殊二叉树

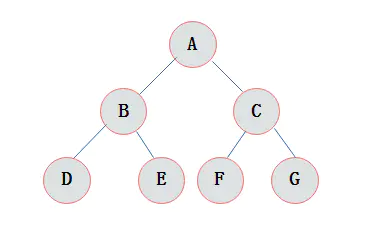
##### 斜树

所有的结点都只有左子树的二叉树叫左斜树。所有结点都是只有右子树的二叉树叫右斜树。这两者统称为斜树。



##### 满二叉树

在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。  
满二叉树的特点有：  
1）叶子只能出现在最下一层。出现在其它层就不可能达成平衡。  
2）非叶子结点的度一定是2。  
3）在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多。

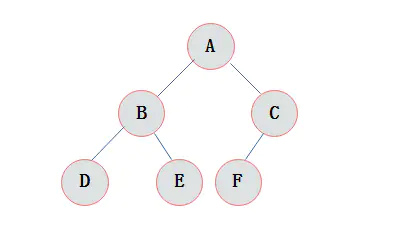


##### 完全二叉树

对一颗具有n个结点的二叉树按层编号，如果编号为i(1<=i<=n)的结点与同样深度的满二叉树中编号为i的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。

图3.5展示一棵完全二叉树。

不知道这个i跟那个i有什么区别？



**特点**：  
1）叶子结点只能出现在最下层和次下层。  
2）最下层的叶子结点集中在树的左部。  
3）倒数第二层若存在叶子结点，一定在右部连续位置。  
4）如果结点度为1，则该结点只有左孩子，即没有右子树。  
5）同样结点数目的二叉树，完全二叉树深度最小。  
**注**：满二叉树一定是完全二叉树，但反过来不一定成立。

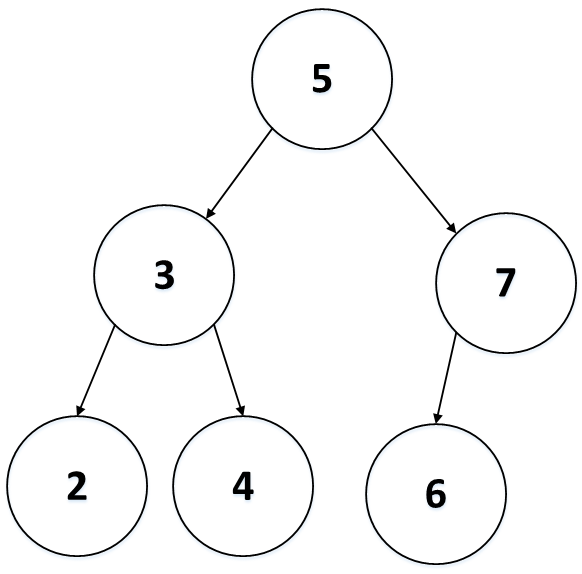
**根据存储数据结构的划分可以划分为二分查找树，平衡二叉树。参考文章**

**https://www.cnblogs.com/sgatbl/p/9426394.html**

##### 二分查找树

性质

二叉查找树实际上是数据域有序的二叉树，即对树上的每个结点，都满足其左子树上所有结点的数据域均小于或等于根结点的数据域，右子树上所有结点的数据域均大于根结点的数据域。

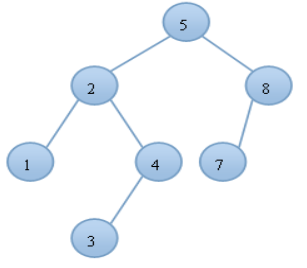


例如：2<3 放入左子树，4>3放入右子树

##### 平衡二叉树

平衡二叉树是由前苏联的两位数学家G.M.Adelse-Velskil和E.M.Landis提出，因此一般也称作AVL树，AVL树本质还是一棵二叉查找树，只是在其基础上增加了“平衡”的要求。所谓平衡是指，对AVL树的任意结点来说，其左子树与右子树的高度之差的绝对值不超过1，其中左子树与右子树的高度因子之差称为平衡因子。

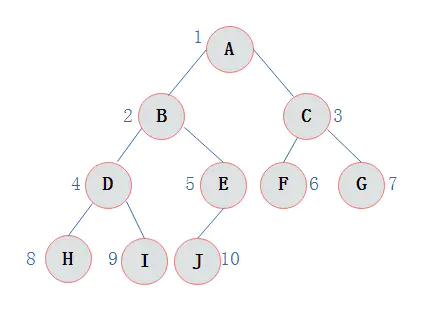
通常来讲就是左子树的高度与右子树的高度差值不能超过1。

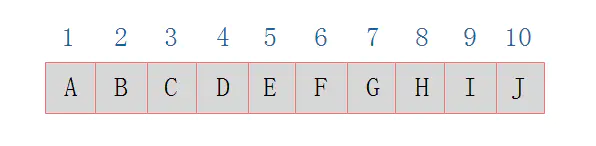


#### 二叉树的存储结构

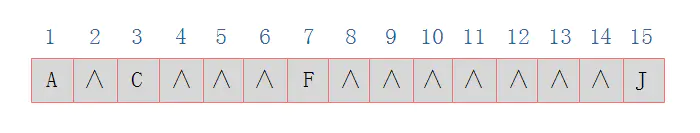
###### 顺序存储

二叉树的顺序存储结构就是使用一维数组存储二叉树中的结点，并且结点的存储位置，就是数组的下标索引





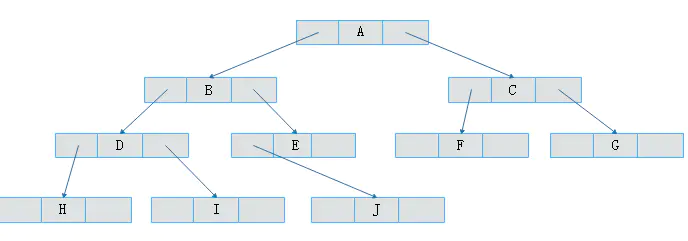
顺序存储适用与完全二叉树，对于其它二叉树结构会存在空间的浪费



###### 二叉链表

二叉树的每个结点最多有两个孩子。因此，可以将结点数据结构定义为一个数据和两个指针域。





### 红黑树

#### 定义

R-B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

#### 性质

1. **每个节点或者是黑色，或者是红色。  
   （2）根节点是黑色。  
   （3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]  
   （4）如果一个节点是红色的，则它的子节点必须是黑色的。  
   （5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**



## 集合

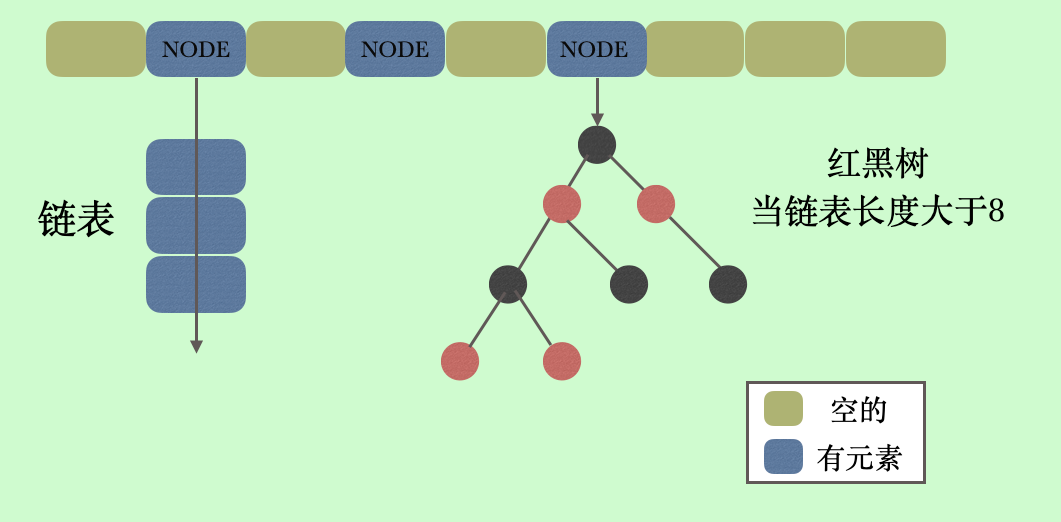
### Map

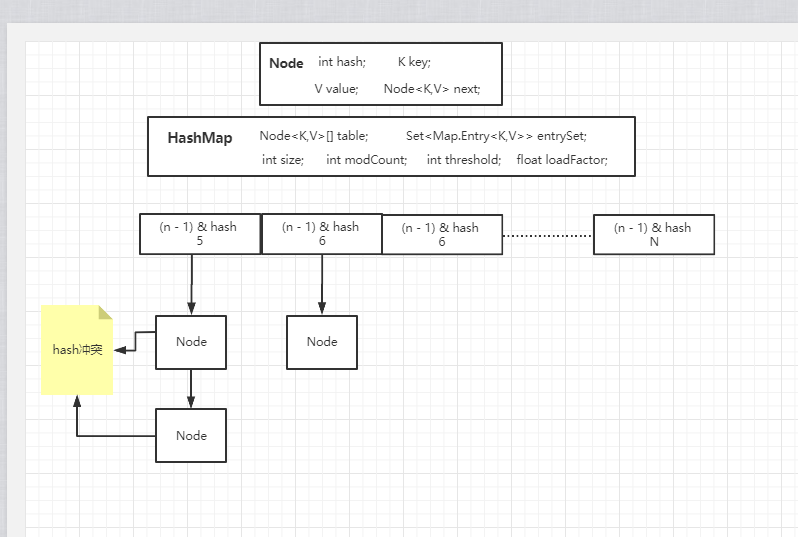
#### HashMap

##### Jdk1.8

###### 存储结构

使用的是数组+链表，红黑树





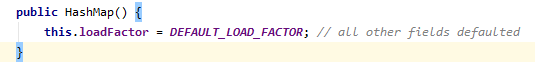
###### 源码分析

HashMap定义属性

**transient** Node<K,V>[] **table**;  
**transient** Set<Map.Entry<K,V>> **entrySet**;  
**transient int size**;  
**transient int modCount**;**int threshold**;  
**final float loadFactor**;

初始化一个map





1. 加载因子赋值为0.75
2. 创建一个空的实例，并不会初始化数组长度

放入数据



**public** V put(K key, V value) {  
 **return** putVal(*hash*(key), key, value, **false**, **true**);  
}

/\*\*

\* onlyIfAbsent如果为true，则不更改现有值

\* evict如果为false，则表处于创建模式

\*/

**final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  
 **boolean** evict) {  
 Node<K,V>[] tab; Node<K,V> p; **int** n, i;  
 **if** ((tab = **table**) == **null** || (n = tab.**length**) == 0)  
 n = (tab = resize()).**length**;  
 **if** ((p = tab[i = (n - 1) & hash]) == **null**)  
 tab[i] = newNode(hash, key, value, **null**);  
 **else** {  
 Node<K,V> e; K k;  
 **if** (p.**hash** == hash &&  
 ((k = p.**key**) == key || (key != **null** && key.equals(k))))  
 e = p;  
 **else if** (p **instanceof** TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);  
 **else** {  
 **for** (**int** binCount = 0; ; ++binCount) {  
 **if** ((e = p.**next**) == **null**) {  
 p.**next** = newNode(hash, key, value, **null**);  
 **if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) *// -1 for 1st* treeifyBin(tab, hash);  
 **break**;  
 }  
 **if** (e.**hash** == hash &&  
 ((k = e.**key**) == key || (key != **null** && key.equals(k))))  
 **break**;  
 p = e;  
 }  
 }  
 **if** (e != **null**) { *// existing mapping for key* V oldValue = e.**value**;  
 **if** (!onlyIfAbsent || oldValue == **null**)  
 e.**value** = value;  
 afterNodeAccess(e);  
 **return** oldValue;  
 }  
 }

}

定义了一些变量

Node<K,V>[] tab; Node<K,V> p; **int** n, i;

给table一个初始值[]

n = (tab = resize()).**length**;

tab = resize();

**final** Node<K,V>[] resize(){

}

Resize()方法第一次放入数据代码提取

int newCap = ***DEFAULT\_INITIAL\_CAPACITY***; 初始容量16  
int newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***); 0.75\*16=12 扩容数

**threshold** = newThr; 12

Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];初始化一个长度为16的Node数组  
**table** = newTab;

**return** newTab; 返回数组，resize方法结束

将数据放入数组

**if** ((p = tab[i = (n - 1) & hash]) == **null**)  
 tab[i] = newNode(hash, key, value, **null**);

++**modCount**;   
**if** (++**size** > **threshold**) 大于12就扩容  
 resize();

afterNodeInsertion(evict); true 一个空方法

put方法总结

判断table是否为空

table 为空

执行resize()方法,给table[]一个初始长度16,负载因子0.75,扩容长度为12

判断tab[i]的是否为空

通过hash算法，算出i,判断该位置是否空数据

tab[i] 等于空

tab[i] = newNode(hash, key, value, **null**);

tab[i] 不等于空

1.判断key是否重复

* 1. key重复

1.1.1直接将value替换掉之前的

* 1. key不重复并且key属于TreeNode

* 1. key不重复

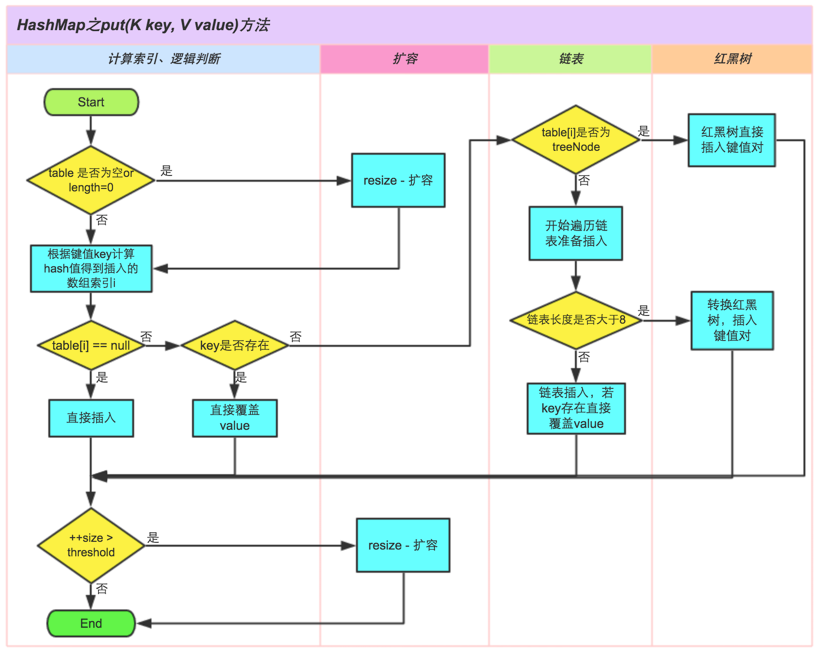
++modCount;

++size

判断是否需要扩容

* 1. 是，扩容 resize();

return null;



## 多线程

### 线程安全问题产生的原因

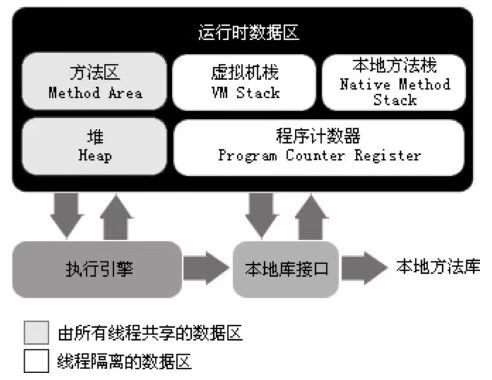
java的主内存与线程私有内存是线程安全问题产生的根本原因

## 设计模式

# Java虚拟机

## java内存模型

### 运行时内存



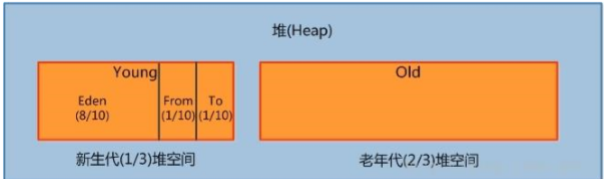
#### 程序计数器

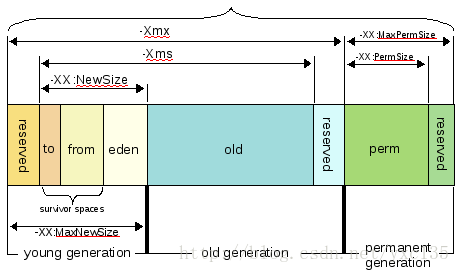
是当前线程所执行的字节码的行号指示器。

#### 堆

是java虚拟机管理内存最大的一块，存放的是对象实例。几乎所有的对象实例都在这里分配内存。

堆从垃圾回收的角度可以分为新生代，老年代。



#### java虚拟机栈

描述的是java方法执行的内存模型。每个方法在执行的时候会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出栈等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈出栈的过程。



##### 栈帧

每个方法执行，都会创建一个栈帧，伴随着方法从创建到执行完成。用于存储局部变量表，操作数栈，动态链接，方法出口等。

###### 局部变量表

主要用于存储方法参数和定义在方法体内的局部变量这些数据类型包括各类基本数据类型、对象引用（reference），以及returnAddressleixing。大小在编译时确定，运行期间不会更改局部变量表大小。

局部变量表中的变量只在当前方法调用中有效

#### 本地方法栈

本地方法栈与java虚拟机栈非常相似，java虚拟机栈为本地方法提供服务，本地方法栈为native方法提供服务。

#### 方法区

它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

永久代是jdk1.7之前方法区的具体实现，元空间是jdk1.8之后方法区的具体实现



###### 永久代

使用的是jvm内存空间。

"java.lang.OutOfMemoryError: PermGen space

###### 元空间

使用的是本地内存。

-XX:MetaspaceSize，初始空间大小

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

元空间的特点

1. 充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致。
2. 每个加载器有专门的存储空间
3. 只进行线性分配
4. 不会单独回收某个类
5. 省掉了GC扫描及压缩的时间
6. 元空间里的对象的位置是固定的
7. 如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉

Java代码测试元空间存储

直接上代码

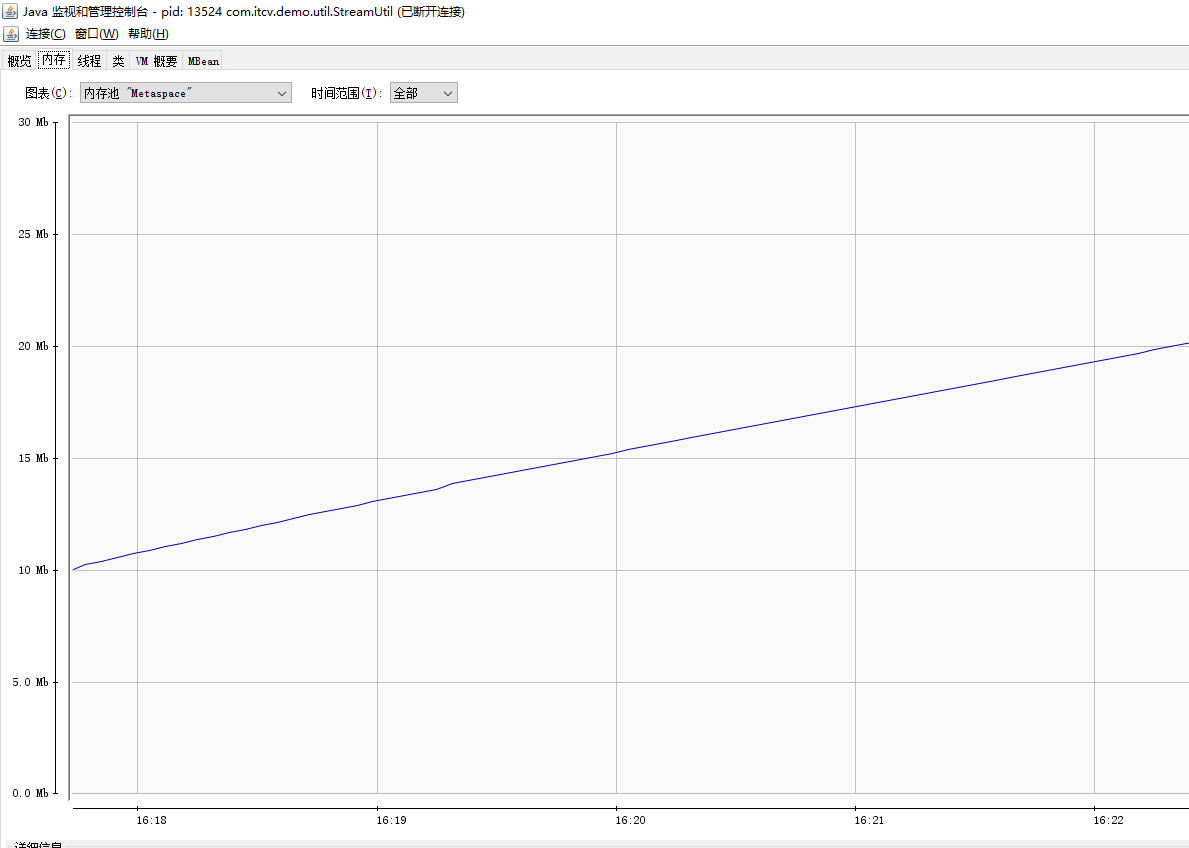
**public class** StreamUtil {  
 **public static void** main(String[] args) {  
 URL url = **null**;  
 List<ClassLoader> classLoaderList = **new** ArrayList<ClassLoader>();  
 **try** {  
 *//获取有关类型加载的JMX接口* ClassLoadingMXBean loadingBean = ManagementFactory.*getClassLoadingMXBean*();  
 url = **new** File(**"E:/data/logs"**).toURI().toURL();  
 URL[] urls = {url};  
 **while** (**true**){  
 ClassLoader loader = **new** URLClassLoader(urls);  
 Class<?> clazz = loader.loadClass(**"Persion"**);  
 clazz.getName();  
 classLoaderList.add(loader);  
 *//显示数量信息（共加载过的类型数目，当前还有效的类型数目，已经被卸载的类型数目）* System.***out***.println(**"total: "**+ loadingBean.getTotalLoadedClassCount());  
 System.***out***.println(**"active: "**+ loadingBean.getLoadedClassCount());  
 System.***out***.println(**"unloaded: "**+ loadingBean.getUnloadedClassCount());  
 Thread.*sleep*(100);  
 *// System.out.println(classLoaderList.size());* }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

实现原理

从外部不停的加载class文件到内存中，查看元空间什么时候奔溃。

元空间最小值为20M,最大值为本机内存。

查看结果



通过jconsole查看内存分配情况

###### 运行时常量池

运行时常量池是方法区的一部分。用于存放编译期生成的各种字面量和符号引用。相对与Class文件常量池的另外一个特征是具备动态性，指：运行期间也可以将新的常量放入池中。

#### 直接内存

避免了在Java堆和Native堆中来回复制数据。

### Java内存分配机制

参考文章

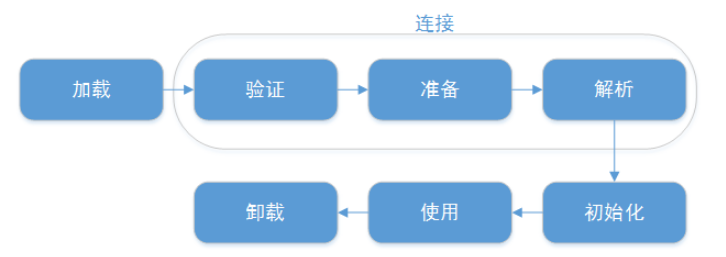
https://blog.csdn.net/freekiteyu/article/details/77992277

## Jvm类加载机制

参考文章

https://www.cnblogs.com/Cubemen/p/10913633.html

### 类加载的过程



类从加载虚拟机内存中开始到卸载出内存为止，生命周期包括：加载、验证、准备、解析、初始化、使用、卸载。

#### 1.加载

把class文件加载到内存中（**方法区**），不一定是文件可以是二进制流

#### 2.验证

##### **文件格式验证**

即验证类文件结构

##### 元数据验证

这个是否有父类，父类是否继承了不允许被继承的类等等

##### 字节码验证

对类的方法体进行校验，JDK1.6后只需检查StackMapTable属性中的记录是否合法，JDK1.7后对于主版本号大于50的Class文件，使用类型检查来完成数据流分析

##### 符号引用验证

全限定名是否能找到对应的类，在指定类中是否存在符合方法的字段描述以及简单名称描述的方法，字段。访问性是否正确。验证不成功会抛出java.lang.incompatibleClassChangeError异常的子类。

#### 3．准备

正式为**类变量**分配内存并设置类变量**初始值**的阶段；

。这个时候进行内存分配的只包括类变量（被static修饰的变量），并不包括实例变量，实例变量是在对象实例化时随对象一起分配在java堆中。

Private static int a =123;初始值为0

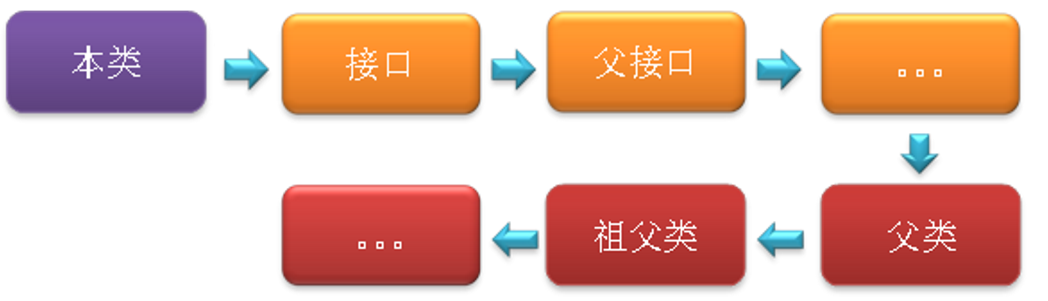
Private finale static int a =123; 初始值为123；

#### 4.解析

将符号引用转换为直接引用的过程。

解析动作主要针对：

1. 类或接口、字段（类成员变量）、类方法、接口方法等引用进行。
2. 类或接口的解析：判断所要转化成的直接引用是对数组类型，还是对普通的对象类型的引用，从而进行不同的解析。
3. 字段解析：对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



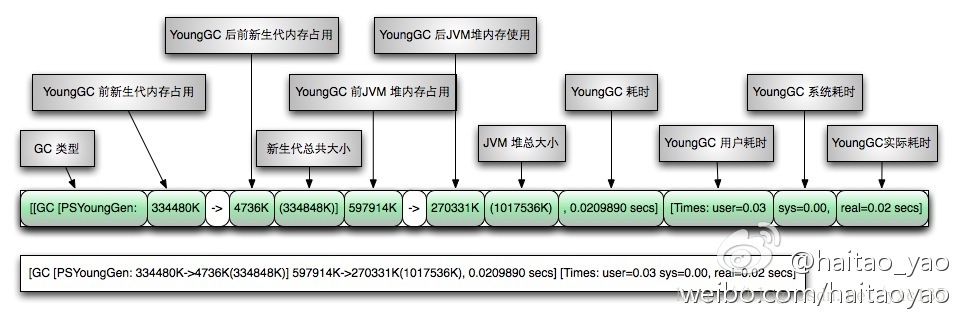
1. 类方法解析：对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。
2. 接口方法解析：与类方法解析步骤类似，由于接口不会有父类，因此，只递归向上搜索父接口就行了。

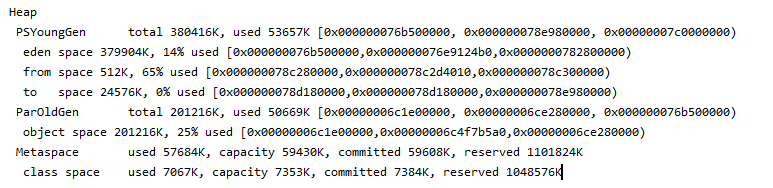
#### 5.初始化

是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码。

### Java中的类加载器与双亲委派机制

## JvmGc日志





# Spring

# SpringMVC

# SpringBoot

# Dubbo

## Dubbo 原理



1. 服务端启动时会把所有接口注册到注册中心
2. 客户端启动时，订阅所需要的服务
3. 订阅内容变更时，会推送订阅的消息
4. 客户端启动时，会与服务端建立长连接，然后进行数据通信
5. 服务端，客户端启动后，后台会启动定时器，发送统计数据给monitor
6. 服务启动的时候，provider和consumer根据配置信息，连接到注册中心register，分别向注册中心注册和订阅服务
7. register根据服务订阅关系，返回provider信息到consumer，同时consumer会把provider信息缓存到本地。如果信息有变更，consumer会收到来自register的推送
8. consumer生成代理对象，同时根据负载均衡策略，选择一台provider，同时定时向monitor记录接口的调用次数和时间信息
9. 拿到代理对象之后，consumer通过代理对象发起接口调用
10. provider收到请求后对数据进行反序列化，然后通过代理调用具体的接口实现

## DubboSpi

### javaSpi

#### javaSpi的机制

是一种将服务接口与服务实现分离以达到解耦、大大提升了程序可扩展性的机制。

#### 实现javaSpi

1. 定义一个接口
2. 在META-INF/services目录下，创建该接口的同名文件
3. 该文件的内容就是接口的具体实现类的全类名（可以是多个）
4. 通过ServiceLoader.load()加载实现类

#### JavaSpi的缺点

1. 不能按需加载,全部加载,浪费资源；
2. 获取某个实现类的方式不够灵活，只能通过遍历；
3. 多个并发多线程使用 ServiceLoader 类的实例是不安全的；
4. 加载不到实现类时抛出并不是真正原因的异常，错误很难定位；

### DubboSpi的增强

1. 对 Dubbo 进行扩展，不需要改动 Dubbo 的源码；
2. 延迟加载，可以一次只加载自己想要加载的扩展实现。
3. 增加了对扩展点 IOC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点
4. Dubbo 的扩展机制能很好的支持第三方 IoC 容器，默认支持 Spring Bean。

### 实现dubboSpi

1. Spi文件存储路径在META-INF\dubbo\internal 目录下 文件名为接口的全路径名 接口的包名+接口名；
2. 每个spi文件里面的格式定义为： 扩展名=具体的类名，例如 dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtoco