# 1 栈内存和堆内存

1.栈内存存放部分基本类型的变量（包括byte，boolean，short，char，int，float，long，double），以及对象的引用（包括对象的引用，String对象的引用，数组对象的引用）。

（注意：栈内存中不存放String基本类型对象，但存放String对象的引用，因为栈内存中存放的数据的大小和生存周期必须是确定的）

1. 堆内存存放所有new创建的对象（比如：new String(“hello”)），以及数组的数据。

## 栈和堆的区别

1. 栈中的数据的大小和生命周期必须是确定的；堆中的数据的大小和生命周期都是运行时动态分配的。
2. 栈的存储速度比堆快（因为堆空间时运行时动态分配的），仅次于CPU寄存器。

# 2 == 和equals的区别

1. ==只有在2个引用类型指向了同一对象时才返回真值。如图1，str1，str2，都存放在栈中，”myString”存放在常量池中，且str1和str2都指向”myString”，因为str1和str2指向同一个”myString”，所以str1==str2。

图1



1. equals只有在对象值相等时才返回真值。如图2，str1，str2存放在栈中，2个不同的new String(“hello”)存放在堆中，然后str1和str2分别指向它们。因为str1和str2指向的对象的值都是”hello”，所以str1.equals(str2)返回真。

图2



# 3 Java值传递和引用传递

Java参数传递只有值传递，当传递的是对象类型时，实际上是创建一个引用的副本，指向该对象，然后传递的是这个引用副本。

如图1，传递到change方法时，实际上创建了一个persion引用的副本，比如叫person1，所以无论change方法里面怎么改变p，实际上都在改变person1，而2个打印的person实际上还是一样的。

图1



# 4 Java进程，JVM，Java内存模型

一个Java进程，对应一个Java虚拟机（JVM），对应一套Java内存模型。在Windows看一个Java进程就是一个javaw.exe，Linux上看就是ps -rf|grep java中的一个Java进程。

# 5 进程和线程的区别

1. 进程拥有系统资源；线程基本不拥有系统资源，它拥有线程的资源，还有独立的程序计数器，一组寄存器和栈。

（注意：一个进程的栈会被多个线程瓜分成多个线程栈）

1. 线程执行开销小，但不利于资源的管理（特别是共享资源）；进程相反。

# 6 线程的状态

图1



图2



（注意：start()后进入的是可运行状态，需要获取CPU时间片才进入运行状态）

（注意：因为wait会释放锁，所以被唤醒之后还需要获得锁才能进入可运行状态）

# 7 hashCode() 哈希值

## 7.1 hashCode()和equals()的关系

hashCode()主要是在哈希表（比如：HashSet，HashMap等）中起作用。

当向哈希表添加对象object时，会用hashCode()方法计算object的哈希值，然后通过哈希值来定位到该对象在哈希表中的位置；如果该位置没有对象，直接放进去，如果有，则用equals()判断是否相等，如果key相等或equals()则覆盖值，如果不相等则...???。

因此，hashCode()相等，只是代表2个对象在哈希表中的位置相等，此时equals()不一定相等。

但如果equals()相等，因为一个对象只能在哈希表中出现一次，所以位置一定相等，所以hashCode()一定相等。

（注意：重写equals()时必须重写hashCode()）

## 7.2 为什么需要hashCode()

因为对象比较时，如果每次都使用equals()，效率很低；而先用hashCode()比较，由于每个类的hashCode()可能有自己的实现，但统一的是，如果hashCode()不一样，就没必要用equals()比较了。

## 7.3 Object的hashCode()

Object的hashCode()就是对象的地址。

图1



## 7.4 String的hashCode()

反正就是一个以31为权的算法，但不同字符串通过这个方式计算出来的结果可能相同。比如，字符串"gdejicbegh"与字符串"hgebcijedg"具有相同的hashCode()返回值-801038016。

图1



# 8 HashMap和HashTable的区别

1. HashMap允许key和value是null（因为HashMap对null作了特殊的处理），HashTable不允许key或value是null。
2. HashMap不是同步的，Hashtable是同步的。

# 9 ArrayList和LinkedList的区别

1. ArrayList的底层是数组，查找某个元素的时间复杂度是O(1)
2. LinkedList每个节点存储了2个引用，一个指向前一个节点，一个指向后一个节点，查找某个元素的时间复杂度为O(n)；但插入到任意位置，或者删除任意位置的时候更快。
3. LinkedList存储更占空间，因为每个节点存储了2个引用。

# 10 Comparable接口和Comparator接口的区别

图1



1. 当一个类可修改时，可以实现Comparable接口，然后就可以对这个类的集合进行排序。

图2



图3



1. 但有时候不是很多类都是可以给我们修改的，或修改代价太高，这时可以不实现Comparable接口，而是使用Comparator接口直接排序。

图1



图2



# 11 Enumeration和Iterator的区别

Iterator：用来遍历Collection集合中的元素。

图1



图2



Enumeration：

图3



区别：

1. Enumeration没有删除方法，只有遍历；Iterator可以删除集合元素。
2. Enumeration是先进后出，Iterator是先进先出。
3. Iterator比Enumeration安全，因为Iterator如果在遍历过程中检测到集合被修改，会抛出ConCurrentModificationException异常。

# 12 内存泄漏

内存泄漏是指一个内存对象的生命周期超出了程序需要它的时间，有时也称为“对象游离”。

换句话说，某个不再使用的对象仍然占用着内存空间。

## 12.1 内存泄漏的场景

原文链接：<http://www.jb51.net/article/92311.htm>

### 12.1.1 静态的集合类引起的内存泄露

o引用虽然设成null，但v中仍然持有每一个Obejct对象的地址，因为Java参数传递是值传递，传递的是每一个Obejct对象的地址。

图1



### 12.1.2 集合里面的对象修改后，remove(obj)不起作用

因为修改后已经不是同一个对象，remove(obj)方法移除不掉，仍然占用着内存。

图1



### 12.1.3 各种连接未释放

各种连接（数据库连接，socket连接，io连接，网络连接），除非显式调用它的close()方法，否则GC不会回收。

### 12.1.4 单例模式持有外部对象的引用

因为单例模式一般是static的单例，它在JVM的整个生命周期存在，如果它持有了一个外部对象的引用（比如ActivityContext），那么这个外部对象就一直不能释放。

（注意：如果传入的是ApplicationContext则不会内存泄漏，因为ApplicationContext的生命周期就是整个应用的生命周期，如图2）

图1



图2



## 12.2 内存泄露和内存溢出的区别

注意：内存泄露和内存溢出是2个概念，内存泄露就是本不该存活的对象还存活着，内存泄露是导致内存溢出的其中一个原因。

# 13 Java设计模式

Java设计模式应符合开闭原则：对类的拓展开放，对类的修改关闭。

## 13.1 策略模式

原文链接：<https://www.cnblogs.com/wuyudong/p/5924223.html>

策略模式：将算法的责任和算法的实现分离，形成抽象策略（比如：Strategy，抽象策略）和具体策略（比如：ConcreStrategyA，具体策略A）。

环境（比如：Context）中保存一个抽象策略的引用（strategy），并提供一个环境的接口（contextInterface）来调用抽象策略的接口（strategyInterface）。

这样，用户代码只需要给环境设置抽象策略引用（strategy）的具体实现，然后调用环境的接口（contextInterface）即可。

图1



### 13.1.1 优点

1.用户代码只需要操纵环境（Context），不需要操纵抽象策略（Strategy）或者具体策略（ConcreStrategy）

2.新增具体策略（ConcreStrategy）时，环境（Context）不需要改动，只要用户代码给环境设置新的抽象策略引用（strategy）的具体实现即可。

（注意：用户代码和环境是不同的概念，用户代码调用环境代码）

### 13.1.2 缺点

1.客户端必须知道所有的具体策略类，并自行决定使用哪个具体策略类。

### 13.1.3 适用场景

1.一个行为（抽象策略），有多种不同的算法实现（具体策略）；系统必须从多种算法实现中选择一种。

### 13.1.4 何为环境，何为抽象策略？

原文链接：<http://blog.csdn.net/zcm101/article/details/9344233>

抽象策略一般是一种行为，比如：说话；那么具体策略就是说中文，说英文，说日文等。

环境一般是具有这种行为的实体（model层，不是service层，model联系一下现实环境中的model），比如：人。环境可能也有一些子类，比如中国人，英国人等。

## 13.2 观察者模式

原文链接：[http://www.cnblogs.com/ZhangHaoShuaiGe/p/7840131.html#3868362](http://www.cnblogs.com/ZhangHaoShuaiGe/p/7840131.html" \l "3868362)

### 13.2.1“推式”观察者模式

定义一个主题接口（Subject），一个观察者接口（Observer）。

主题接口有3个方法：注册观察者，删除观察者，通知观察者；以及一个观察者列表的引用。

观察者接口只有1个方法：被通知（update()），主题通知观察者时调用此方法；以及一个主题对象的引用。

图1



### 13.2.2“Java内置的”观察者模式

Java内置了一个可观察者类（Observable，相当于上面的主题接口Subject），一个观察者接口（Observer）。

具体主题实现可观察者类（Observable），不需要维护观察者列表的引用，因为Observable已经有维护了这个列表。

图1



具体观察者实现Observer接口。

图2



Java内置的观察者模式的缺点：

1. 可观察类（Observable）是一个类，不是一个接口，如果某类想同时具有可观察类（Observable）和另一个类的行为，会陷入两难。

### 13.2.3 使用PropertyChangeSupport和PropertyChangeListener的观察者模式

PropertyChangeListener监听bean的属性；当bean属性发生变化时，PropertyChangeSupport的firePropertyChange方法将一个事件发送给所有已经注册的监听器PropertyChangeListener。

PropertyChangeSupport类似于主题接口（Subject），PropertyChangeListener类似于观察者接口（Observer）。

具体主题可以不继承，维护一个PropertyChangeSupport的引用，实现主题的注册观察者，删除观察者和通知观察者的方法，底层调用PropertyChangeSupport。

图1



具体观察者实现PropertyChangeListener。

（注意：看图2直到具体观察者ChangeListener没有指定监听Mybean的哪一个属性，说明Mybean的任意属性改变都会通知ChangeListener）

图2



## 13.3 装饰者模式

原文链接：<http://www.cnblogs.com/ZhangHaoShuaiGe/p/7866610.html>

<https://www.cnblogs.com/jiqing9006/p/5962431.html>

装饰者模式：抽象组件（Component）有抽象组件的方法（methodA()，methodB()等）；

具体组件（ConcreteComponent）继承抽象组件（Component）。

装饰器（Decorator）是用来装饰抽象组件（Component）的。

（注意：为什么需要装饰器，而不是再定义一个子类继承具体组件呢？原因是装饰器装饰的是抽象组件（Component），因为所有的具体组件（ConcreteComponent）继承抽象组件（Component），所以装饰器可以装饰所有的具体组件。但，如果是定义子类继承其中一个具体组件，那这个子类装饰的具体组件只能是其中一个）

图1



（注意：装饰器为什么需要继承抽象组件（Component）？原因是调用组件方法时，只需要一直维持一个抽象组件（Component）的引用，就可以调用装饰器的方法，如图2引用的类型一直是抽象组件（Component））

图2



### 13.3.1 何为抽象组件？何为装饰器？

抽象组件就是被装饰对象（可以理解为1个主体，比如：咖啡）；装饰器就是装饰上去的东西（可以理解为配料，比如：牛奶，巧克力，糖；配料，永远不是主体）。而被装饰对象会有自己的不同实现，装饰器也有自己不同的实现。

## 13.4 单例模式

原文链接：<http://www.cnblogs.com/ZhangHaoShuaiGe/p/7885413.html>

### 13.4.1 使用场景

有些情况下，某些类比如：缓存，线程池等，只需要有1个实例，如果有多个实例，会造成很多问题。

### 13.4.2 最简单的单例

1.将构造函数私有化，这样无法通过构造函数new多个实例。

2.定义1个公开的（public），静态（static）的方法（getInstance()）构造实例。

（注意：如果不是public，那这个方法外部调用不了，1个实例都构造不出来；如果不是static，需要实例才能调用此方法，那和使用这个方法构造实例自相矛盾）

3.getInstance()方法判断私有的（private），静态的（static）变量是否唯一，决定是否创建实例。

图1



缺点：1.多线程并发时会构造多个实例。

### 13.4.3 双重加锁的单例

注意：一个synchronized加锁很好理解，之所以synchronized外面加一个if判断是为了提高效率，试想当第三次之后调用getInstance()方法时，此时Myclass对象肯定已经new出来了，没必要浪费时间去加锁，直接第一层的if就已经返回这个实例了。

图1



缺点：1.重排序，因为JVM创建对象分为三个过程：（1）分配对象的内存地址，比如：123 （2）在该内存地址123构造对象（3）将该内存地址123赋值给引用变量。在myClass=new MyClass()中，由于重排序，（3）可能发生在（2）前面，此时myClass已经有值，但MyClass对象可能构造了一半，还没构造完；当f3调用getInstance()方法，第一个if判断返回了一个未构造完全的myClass引用。

（注意：这个内存地址123在（2）（3）操作都用到，但都是读，所以没有数据依赖性，是可能出现重排序的）

可以用volatile禁止这个Class变量的重排序解决重排序导致的问题。

图3



### 13.4.4 饿汉式的单例

饿汉式就是上来就创建对象。

图1



## 13.5 适配器模式

原文链接：<http://www.cnblogs.com/ZhangHaoShuaiGe/p/7911221.html>

### 13.5.1 对象适配器

对象适配器：

有一个面向客户的目标接口（Target），还有一个待适配的接口（Adaptee）。 客户想用到Adaptee的功能，但它只能看到Target。

这时候就需要定义一个适配器（Adapter），实现Target，并维护一个Adaptee的引用。并且在Adapter的request()方法中调用Adaptee的specificRequest()方法。

（注意：客户真正想用的是待适配的接口（Adaptee），可惜他看到的是目标接口（Target））

图1



### 13.5.2 类适配器

原文链接：<http://blog.csdn.net/yujin753/article/details/46287643>

类适配器：有一个面向客户的目标接口（Target），还有一个待适配的接口（Adaptee）。 客户想用到Adaptee的功能，但它只能看到Target。

这时候就需要定义一个适配器（Adapter），实现Target，同时实现Adaptee。

并且在Adapter的request()方法中调用Adaptee的specificRequest()方法。

（注意：客户真正想用的是待适配的接口（Adaptee），可惜他看到的是目标接口（Target））

图1



### 13.5.3 对象适配器和类适配器的区别

1.类适配器使用继承，对象适配器使用组合，根据设计模式的多用组合，少用继承的原则，对象适配器更优。

2.类适配器使用继承Adaptee的方式，所以不能适配Adaptee的子类；但对象适配器使用组合的方式，可以适配Adaptee的子类。

### 13.5.4 适用场景

客户看到的只有A接口，却想调用B接口的方法，但A和B接口没有直接的关联关系。

# 14 short s1 = 1；是否需要强转

不需要，像short s1 = 1; char c1 = 3;byte b1 = 50;这些在目标有效值范围内的，都不需要强转；但像short s1 = 65525;这种超出了short有效值范围的，需要强转，否则会编译错误。

# 15 Java 字面值

字面值就是固定的值。

## 15.1 字面值分配

1.整型：比如10，12。

2.浮点型：这里包括float和double，比如：11.4F，10D，double a = 10中的10等。

3.字符型：比如：’c’和’1’等。

4.字符串型：比如：”abc”。

5.布尔型：只有true和false。

6.特殊字面值：包括null和class Iiteral（比如：Integer.class，输出发现是java.lang.Integer）

## 15.2 字面值 内存分析

字面值是存放在方法区的常量池中。

# 16 String，StringBuffer，StringBuider的区别

1. String变量的对象值是不可变的，当它的对象值发生改变后，String变量指向了新的对象；StringBuffer和StringBuider变量的对象值是可变的，当它的对象值发生改变后，String变量还是指向原来的对象。
2. StringBuffer很多方法有synchronized关键字，所以是线程安全的，但速度慢。StringBuilder线程不安全，但速度快。

# 17 嵌套类，内部类，静态嵌套类

嵌套类：在一个类里面再定义个类，外面的类OuterClass叫外围类，里面的类NestedClass叫嵌套类。

图1



嵌套类根据是否静态分为：静态嵌套类和非静态嵌套类（非静态嵌套类通常称为内部类）

图2



## 17.1 内部类和静态嵌套类的区别

1.内部类创建前需要先创建外围类，因为内部类的创建包括2个步骤：创建内部类实例，然后将内部类实例绑定某个外围类实例。

图1



而静态嵌套类的创建不需要先创建外围类，外围类更像是它的命名空间而已。

图2



# 18 对象克隆

原文链接：<https://www.cnblogs.com/Qian123/p/5710533.html>

## 18.1 引用复制

引用复制不是对象的克隆，因为引用复制的结果只有1个对象。

图1



图2



## 18.2 重新new一个对象和克隆的区别

1.new出来的对象都是初始属性值；而克隆的对象可以保存当前属性值。

2.new一个对象，一个一个属性设置麻烦；克隆方法是native方法，速度更快。

## 18.3 浅克隆和深克隆的区别

1.浅克隆对引用类型成员变量复制的是引用地址。

## 18.4 浅克隆

1.被复制的类实现Clonenable接口

（注意：不实现此接口再调用clone()方法会抛出CloneNotSupportedException会异常）

1. 覆盖clone()方法，修饰符改成public（为了能在外部调用clone()方法），方法中调用super.clone()。

图1



## 18.5 深克隆

# 19 数据库隔离级别

## 19.1 脏读，不可重复读，幻读的区别

脏读：事务T1读取了事务T2已经写入到数据库内存，但未提交的数据。如果T2回滚，T1读取的是脏数据。

不可重复读：T1按一定条件读取数据库，T2修改了该查询条件的数据库记录，T1再读取发现结果不一样，这就是不可重复读。

幻读：T1按一定条件读取数据库，T2新增或删除了该查询条件的数据库记录，T1在读取发现记录数不一样，这就是幻读。

（注意：不可重复读强调修改，幻读强调新增或删除）

## 19.2 隔离级别

1.Read uncommitted（读未提交的数据）：就是脏读的意思，不可以防止脏读，不可重复读和幻读。

2.Read committed（读已提交的数据）：防止了脏读，不可以防止不可重复读和幻读。

3.Repeatable read（可重复读）：防止了脏读，不可重复读，不能防止幻读。

4.Serializable（串行化）：防止了脏读，不可重复读和幻读。

注意：MySQL默认级别是Repeatable read（可重复读），Oracle默认级别是Read committed（读已提交的数据）。Oracle只支持Read committed（读已提交的数据）和Serializable（串行化）。

# 20 代理

原文链接：<https://www.cnblogs.com/hadoop-dev/p/7095464.html>

代理就是：代理类代替委托类去做事情。

优点：

1. 隐藏委托类的实现。
2. 客户与委托类解耦，客户只看见代理类。
3. 在不修改委托类的接口实现前提下，在代理类中对委托类接口功能做拓展（比如：前拦截，后拦截等）

## 20.1 静态代理

静态代理：代理类在程序运行前就已经存在。

代理类和委托类都实现同一个委托接口，然后代理类持有一个委托接口的引用。（其实就是策略模式，代理类就是环境类，委托接口和委托类就是抽象算法和具体算法）

图1



图2



缺点：

1. 当有10个不同的委托接口（注意：是委托接口都不一样，不是同一个委托接口不同委托类实现），而代理类的拓展逻辑都是相同的时候，需要为每一个委托接口都写一个静态代理类，造成代理类的膨胀。

## 20.2 JDK动态代理

动态代理：运行时动态生成代理类。

定义一个中介类实现InvocationHandler接口。中介类底层调用委托接口的方法，然后通过中介类动态生成代理类。运行时，代理类的方法执行会被拦截，调用委托接口的逻辑。

图1



图2



缺点：

1. 如果没有委托接口，只有委托类，那么无法实现JDK动态代理。

## 20.3 cglib动态代理

原文链接：<https://www.cnblogs.com/ygj0930/p/6542259.html>

定义一个中介类实现MethodInterceptor接口。中介类底层调用委托接口的方法，然后通过中介类动态生成代理类。运行时，代理类的方法执行会被拦截，调用委托接口的逻辑。

图1



图2



缺点：

1.委托类不能是final修饰的，否则无法继承。

# equals()

## 21.1 Object的equals()

Object的equals()判断是否是同一个对象。所以一般会重写，用来判断对象的内容是否相同。

图1



## 21.2 String的equals()

String的equals()比较对象的内容是否相同。

图1



# Cookie和Session的区别

1. Cookie数据存放在客户端；Session数据存放在服务端。
2. Cookie是所有用户都能看到的；Session是单用户才能看到的。
3. Cookie的数据就算加密也有被解密的风险；Session数据保存在服务器，安全性更高。
4. Cookie如果不设置过期时间，生命周期是浏览器关闭后销毁，如果设置了生命周期，则过期或用户主动清除销毁；Session生命周期是访问JSP或Servlet时创建，默认20分钟内该用户没请求自动销毁，容器关闭也会销毁。
5. 不同浏览器对Cookie的大小和个数有限制。（域指的是域名）

图1



# 序列化

## 23.1 序列化的场景

1. 网络上传输对象。
2. 保存对象的当前状态，可以序列化保存到文件中。
3. 深克隆。

# 24 Java作用域

default：子类不能访问。

protected：同包可以访问。

## 25 JSP内置对象

原文链接：<https://www.cnblogs.com/leirenyuan/p/6016063.html>

# 26 算法的时间复杂度

原文链接：<https://blog.csdn.net/qq_14961511/article/details/39804677>

## 26.1 时间频度（语句执行次数）

一个算法花费具体多少时间我们没必要知道（比如知道执行多少秒），我们只要知道哪个时间执行时间长，哪个算法执行时间短就可以了。

而算法执行时间的长短决定于语句的执行次数，语句的执行次数也称为语句频度或时间频度，记为T(n)，其中n为问题的规模，一般规模n越大，语句执行次数越多，但也有可能是固定的常量。

（注意：for循环里面的语句的执行次数一般等于for循环中的执行次数，如果有多个for，那就是多个for的执行次数的乘积，如图1中语句2的执行次数是(n-1)\*(2n+1)）

图1



## 26.2 时间复杂度

如果有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于0的长度，则称f(n)是T(n)的同数量级函数，它们的关系表示为T(n)=O(f(n))。

（注意：一般来说，f(n)取T(n)中n的最高阶）

而O(f(n))就是该算法的渐进时间复杂度，简称时间复杂度。

（换句话说：O(f(n))能够表示当规模n变大时，语句执行次数的增长趋势，如果是常量，则在xy坐标轴上是一条直线，0增长）

## 26.3 常见的时间复杂度

如图1是常见的时间复杂度。

（注意：指数阶劣于k次方阶）

图1



## 26.4 时间复杂度的计算

一般只用看当规模是n时，统计每个语句的次数相加得到T(n)，然后取T(n)中n的最高阶得到f(n)，时间复杂度就是O(f(n))。

图1



## 26.5 数组的时间复杂度

# 27 HashMap的实现

原文链接：<https://www.cnblogs.com/chengxiao/p/6059914.html>

hashMap采用数组+链表的实现方式，数组中的每个元素都是一个Enrty对象，每个Entry对象有key，value，哈希值hash，也有指向下一个Entry的指针next。

（注意：不同的key可能有相同的hash值，造成哈希冲突，此时新的元素会链在该位置上一个元素的后面。）

图1



图2



# 28 分布式和集群的区别

单机版：一个项目的所有业务都放在这个项目中，这个项目部署到一台服务器上。

集群：一个项目的所有业务都放在这个项目中，这个项目部署到多台服务器上，通过负载均衡服务器来将请求分摊到多个服务器节点。

分布式：一个项目的各个子业务放在不同的子业务系统中，各个子业务系统独立运行在不同的服务器上，它们之前通过RPC通信。

图1



# 29 PO和VO的区别

* + - 1. PO（Persistant Object）是持久化对象，用于持久化；VO（value Object）是值对象，用于业务层。
      2. PO属性与数据库字段对应；VO根据业务需要定义属性。

# 30 apache Bean.copyProperties(obj1,obj2)

apache Bean.copyProperties(obj1,obj2)使用反射机制将后面的对象的属性值赋值给前面的对象的属性（后赋前）。

（注意：属性名称相同才可以赋值）

## 30.1 优缺点

优点：

1.省去了大量的get/set代码。

缺点：

1. Bean.copyProperties的效率远低于普通的get，set

## 30.2 apache和Spring的Bean.copyProperties(obj1,obj2)的区别

apache是后赋前

Spring是前赋后

图1



# 31 LinkedHashMap ？？？

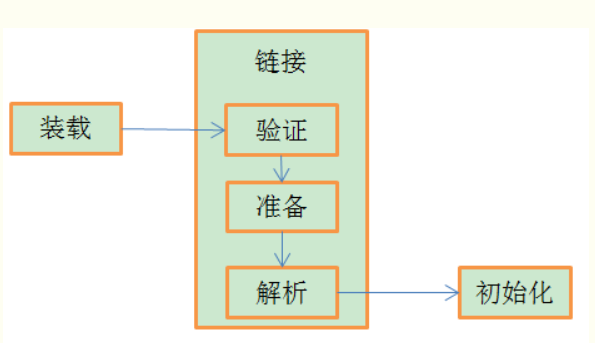
原文链接：<https://blog.csdn.net/justloveyou_/article/details/71713781>

# 32 类加载机制

原文链接：<http://www.importnew.com/25295.html>

JVM类加载机制分为3个部分：装载，链接（连接），初始化。链接又分为3部分：验证，准备，解析。

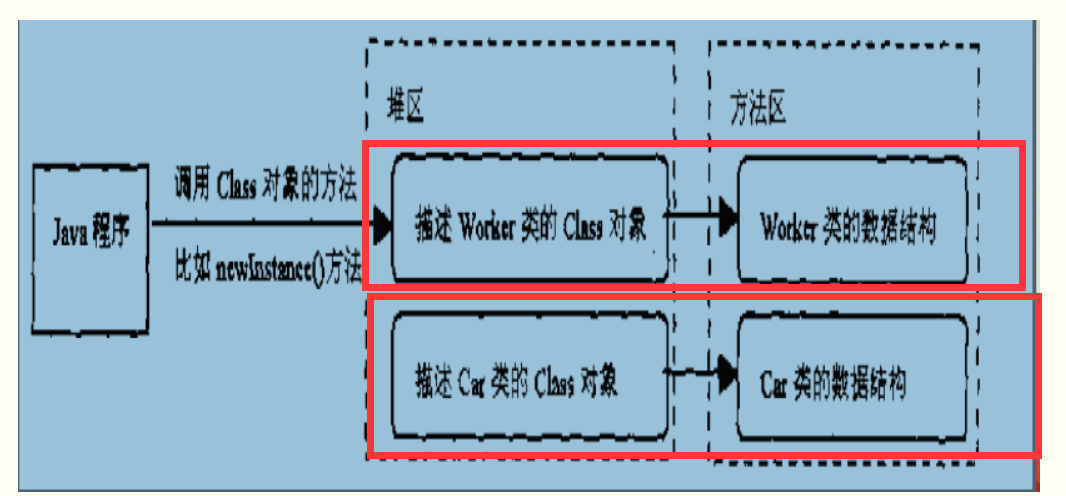
图1



装载：

1. 根据类的全限定名，查找该类的二进制字节流（\*.class文件）。
2. 将这个字节流代表的静态存储结构转化为方法区中的运行时数据结构。
3. 在内存中（一般在堆中）生成一个代表这个类的java.lang.Class对象，作为方法区中这个类的各种数据的访问入口。

图2



验证：

确保\*.class文件中的二进制字节流是否符合当前虚拟机的要求，不会危害虚拟机自身的安全。

准备：

在方法区中为类变量分配内存，并设置类变量的初始值。

（注意：准备阶段不处理实例变量）

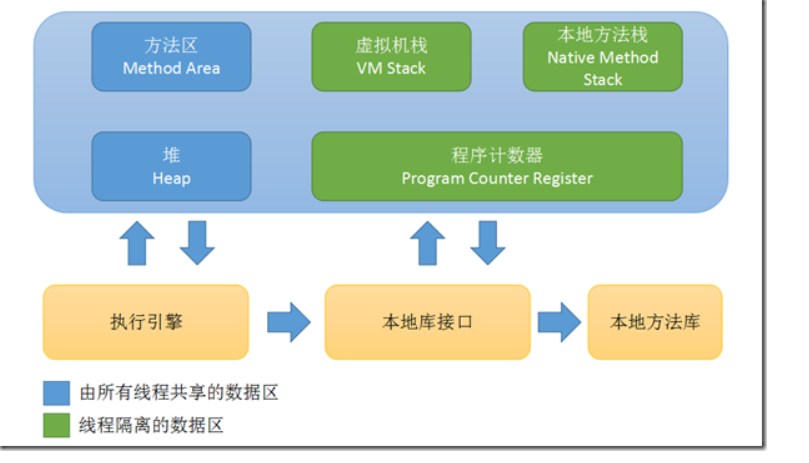
（注意：如果public static int value=123，在准备阶段过后，value的初始值是0，不是123，因为这时候还未执行任何Java方法；但如果是final修饰的，那准备阶段过后，value的初始值是123）

解析：将类中的符号引用转换成直接引用。

初始化：为类变量赋予正确的初始值。可以是声明时指定的初始值，也可以是静态代码块中指定的初始值。

# 34 Java运行时数据区域

图1



## 34.1 程序计数器

程序计数器是线程私有的比较小的区域，记录了该线程正在执行的虚拟机字节码指令的地址。

（注意：分支，循环，跳转，异常处理，线程恢复等功能都依赖这个计数器完成）

### 34.1.1 线程切换依赖程序计数器

一个处理器（对于多核处理器来说是一个内核），在同一时刻，只会执行一条线程中的指令。为了线程切换后，仍然能恢复到以前正确的执行位置，就要靠程序计数器中记录的该线程正在执行的指令的地址。

## 34.2 虚拟机栈

虚拟机栈也是线程私有的区域，线程每执行一个方法，都会创建一个帧栈（帧栈用来存储局部变量表，操作数栈，动态链接，方法出口等信息）入虚拟机栈，方法执行完帧栈出虚拟机栈。

（注意：平时说的栈指的就是虚拟机栈）

### 34.2.1 局部变量表

局部变量表中存放了编译期可知的各种基本类型，还有对象引用和returnAddress（一条字节码指令的地址）。

### 34.2.2 栈异常

StackOverflowError：

OutOfMemoryError

1.当线程太多时，因为每个线程的线程栈，所以会内存溢出，抛出OutOfMemoryError。

## 34.3 本地方法栈

本地方法栈和虚拟机栈作用很相似，不同的是虚拟机栈是执行Java方法，本地方法栈是执行Native方法。

同样会抛出StackOverflowError和OutOfMemoryError。

## 34.4 堆

堆是所有线程共享的内存中最大的一块区域，几乎所有的对象实例都在堆中。

（注意：几乎，但不是所有对象，有一些优化技术会导致对象不在堆中）

### 34.4.1 堆可以分配在物理上不连续的内存空间

堆可以分配在物理上不连续的内存空间。

### 34.4.2 堆异常

OutOfMemoryError：当堆没有内存完成实例的分配，会去拓展堆空间，当堆空间无法扩展时，会抛出OutOfMemoryError。

图1



### 34.4.3 堆的细分 ？？？

堆分为新生代Young和老年代Old。新生代Young进一步被分为Eden和Survivor，Survivor由FromSpace（Survivor1）和ToSpace（Survivor2）组成。

（注意：Survivor是新生代的）

新生代Young的Eden：新创建的对象都在新生代的Eden分配内存，如过Eden空间不足，会触发Minor GC，将Eden存活的对象转入Survivor。

### 34.4.5 指针碰撞 多线程同时分配对象内存

当线程f1为对象A分配内存，指针还来不及移动的时候，f2为对象B分配内存，使用了原来的指针来分配内存，这样会出问题。

法1：虚拟机采用CAS配上失败重试的方法保证指针更新的原子性。

法2：每个线程在堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。然后每个线程在自己的TLAB上分配，只有当TLAB用完需要分配新的TLAB时，才需要同步锁定。

（注意：TLAB也在堆中）

## 34.5 方法区

方法区也是所有线程共享的内存区域，用来存储加载的类信息，常量，字面值，静态变量，即时编译器便以后的代码等数据。

（注意：方法区有一个别名是非堆。）

### 34.5.1 方法区可以分配在物理上不连续的内存空间

方法区可以分配在物理上不连续的内存空间。

### 34.5.2 方法区异常

OutOfMemoryError：当方法区没有内存完成分配时，会去拓展方法区空间，当方法区空间无法扩展时，会抛出OutOfMemoryError。

1.通过CGLIB动态生成大量的增强类，而类信息是保存在方法区中的，所以出现OutOfMemoryError。

### 34.5.3 运行时常量池

运行时常量池是方法区的一部分，存放常量。

## 34.6 直接内存

直接内存不是运行时内存区域的一部分，比如NIO允许直接分配直接内存，然后用堆中的DirectByteBuf对象指向这块内存。

（注意：这个区域的大小会受本机总内存限制，当各个内存区总和大于本机总内存，会抛出OutOfMemoryError。）

### 34.6.1 直接内存异常

OutOfMemoryError：当使用DirectByteBuf对象分配直接内存时，如果直接内存空间不足，会抛出DirectByteBuf。

# 35 垃圾回收器

## 35.1 需要回收的区域

在内存模型中，由3块区域是不需要回收的：程序计数器，JVM栈，本地方法栈。因为它们会随着线程的销毁自动释放。所以只有堆和方法区需要GC。

## 35.2 垃圾回收算法

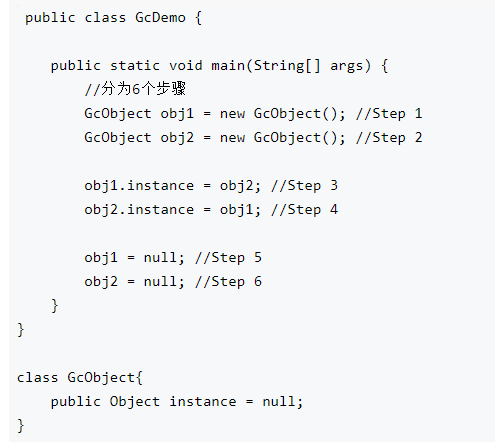
### 35.2.1 引用计数器算法

这个对象每增加一个引用，该对象的引用计数就+1，当该对象的引用计数变为0，则该对象可以当作可回收的目标。

（注意：只是成为可回收目标，不是立刻回收）

缺点：不能解决循环引用的问题。

图1（执行完Step 6后，有2个new GcObject的对象的引用都为1，按照引用计数器的算法不会回收）



### 35.2.2 可达性分析算法（根搜索算法）

以一些可以作为“GC Roots”的对象作为起始点，从这些节点往下搜索，搜索过的路径称为引用链。当一个对象到GC Roots没有任何引用链，说明这些对象到GC Roots是不可达的，不可达的对象被认为是可回收的对象。

（注意：只是成为可回收目标，不是立刻回收）

可以作为“GC Roots”的对象有：

1. 虚拟机栈中的栈帧的局部变量表引用的对象。
2. 本地方法栈中JNI引用的对象。
3. 方法区中类静态属性和常量引用的对象。

缺点：

1. 分析过程需要GC停顿（即STOP The World，停顿所有线程）。

### 35.2.3 标记-清除算法

先标记出所有可回收的对象，标记完成后统一回收被标记的对象。

缺点：

1. 标记和清除2个过程的效率都不高。
2. 标记清除后产生大量不连续的内存碎片，不利于后面分配较大对象。

### 35.2.4 复制算法

将可用内存划分成大小相等的2块，每次只使用其中的一块（整个内存切一半），当这一块用完后，将还存活的对象复制到另一块上，然后将已使用的那一块一次清理掉。

（通常Eden（80%）和Survivor1（10%）占其中一块，Survivor2（10%）占另一块）

优点：

1. 不会出现不连续的内存碎片
2. 运行高效
3. 适合新生代，因为新生代每次回收都有大批对象死去，只有少量存活，而复制算法只复制存活对象，所以此时的复制成本很低。

缺点：

1. 将内存缩小为原来的一半

### 35.2.5 标记-整理算法

先标记出所有可回收的对象，标记完成后将还存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

优点：

1.不会出现不连续的内存碎片

2.不会将内存缩小为原来的一半

### 35.2.6 分代收集算法

对堆中不同代使用不同的收集算法。对于新生代，因为每次都只有少量存活，所以采用复制算法；对于老年代，因为对象存活率高，不适合用复制算法，所以用标记-清除或标记-整理算法。

## 35.3 二次标记

垃圾回收算法会产生可回收的对象，但可回收对象是处于“缓刑”阶段（缓刑阶段属于第一次标记），需要经历二次标记过程才会“处死”。

如果缓刑对象没有覆盖finalize方法，或已经执行过一次finalize方法，则下次回收时会被处死。

（注意：已经执行过一次finalize方法的对象下次回收会被处死，防止对象可以一直通过finalize方法自救而得到永生）

如果缓刑对象覆盖了finalize方法，会将对象放入一个F-Queue队列。然后虚拟机建立一个低优先级的Finalizer线程去调用F-Queue队列中的对象的finalize方法，但不等待这个方法执行完成，防止某对象在finalize方法执行缓慢或死循环导致队列其他对象永久等待。如果对象在finalize方法中拯救自己（比如将this赋值给某个变量），则这个对象会移除出F-Queue队列，否则会真正的被处死。

## 35.4 方法区回收

方法区回收就是对“废弃的常量”和“废弃的类”进行回收。

废弃的常量：如果一个常量池中的常量比如：”abc”没有被任String对象引用，则它是废弃的常量。

废弃的类：同时满足3个条件才是废弃的类：

1. 该类所有的实例对象都已经被回收，即堆中不存在该类的任何实例。
2. 加载该类的ClassLoader已经被回收。
3. 该类对应的java.lang.Class对象没有被引用，即无法通过反射访问该类。

（注意：“废弃的类”可以被回收，但不是必然会回收）

## 35.5 Minor GC

场景：当Eden区或Survivor From区空间不足时，会触发Minor GC，Minor GC采用复制算法。

Minor GC（新生代GC）：

1. 将Eden区和Survivor From区的存活对象复制到Survivor To中，对象的年龄+1
2. 清空Eden区和Survivor From区的对象，交换Survivor From和Survivor To
3. 如果对象的年龄增加到一定年龄（由MaxTenuring决定，默认15岁），就会晋升到老年代。

（注意：Survivor中的对象如果>=Survivor对象的平均年龄，也可以直接进入老年代，不需要等到MaxTenuring的阈值）

## 35.6 分配担保

在发生Minor GC之前，会检查老年代的最大可用连续空间是否大于新生代所有对象总空间，或者检查老年代的最大可用连续空间是否大于历代晋升到老年代对象容量的平均大小，如果大于，可以将Survivor无法容纳的对象直接进入老年代。

## 35.8 Major GC和Full GC

Major GC表示清理老年代，Full GC表示新生代和老年代一起清理。

Major GC：

场景：当老年代空间不足时，会触发Major GC，Major GC采用标记-清除，或标记-整理算法。

1. 扫描老年代，标记存活对象，清除没有标记的对象。

## 35.9 System.gc()

System.gc()其实就是执行了一次Full GC。

## 35.10 GC overhead limit exceeded

原文链接：<https://www.jianshu.com/p/2fdee831ed03>

GC overhead limit exceeded提示你的应用程序花了太多时间去进行垃圾回收，然而并没有什么卵用。默认是超过98%的时间去做GC但只回收了不到2%的内存则会抛出这个错误。

图1



# 36 对象的创建过程

1. 遇到一条new指令时，寻找常量池是否能定位到一个类的符号引用？？？，然后检查这个符号引用代表的类是否已经加载，解析和初始化。

（注意：类加载完成后，对象所需的内存大小可以完全确定）

1. 选择指针碰撞或空闲列表的方式为对象分配所需要的内存。
2. 将分配到的内存空间初始化为零值。

（注意：初始化为零值保证了对象的实例字段在不赋予初始值就可以直接使用而不报错）

1. 对对象进行必要的设置。比如对象头的设置：这个对象是哪个类的实例，怎样找到类的元数据信息，对象的哈希码，对象的GC分代年龄等。
2. 执行对象的<init>方法，为对象的字段设置程序员意愿的值，

### 36.1 指针碰撞和空闲列表

当在堆中为一个对象分配内存时，对象所需的内存大小在类加载后就可以完全确定。

堆中的内存分为规整和不规整：

规整：所有用过的内存放在一边，所有空闲的内存放在另一边，中间放了一个指针作为分界点，那么分配该对象的内存就是指针向空闲空间那边移动一段与对象大小相等的距离，这种方式成为指针碰撞。

不规整：不规整就是用过的内存和空闲的内存相互交错，这样就没办法用指针碰撞，因为指针向空闲空间移动一段距离后，空闲空间不全是空闲的，已使用和空闲内存时交错存在的。这时候需要维护一份空闲列表，记录哪些内存是已使用的，哪些内存是空闲的。分配时空闲列表中选取一块足够大的空间分配给该对象，这种方式称为空闲列表。

堆内存是否规整是由采用的垃圾回收器是否带有压缩整理功能决定的。在使用Serial，ParNew等待Compact过程的收集器，采用的是指针碰撞；在使用CMS这种给予Mark-Sweep算法的收集器时，通常采用空闲列表。

### 36.2 内存空间初始化为零值

不管是通过指针碰撞还是空闲列表，当该对象成功分配了堆内存后，虚拟机会为分配到的内存空间都初始化为零值。这样保证了对象的实例字段在不赋予初始值就可以直接使用而不报错，因为被初始化了零值。

# 37 对象的组成

对象在堆中的内存由3部分组成：对象头，实例数据和对齐填充。

## 37.1 对象头

对象头由2部分组成：第一部分是对象的运行时数据，包括哈希码，GC分代年林，锁状态标志，线程持有的锁，偏向线程ID，偏向时间戳等；第二部分是类型指针，指向它的类元数据。

## 37.2 实例数据

无论是从父类继承的，还是子类中定义中的，都有保存起来。

## 37.3 对齐填充

因为HotSpot虚拟机要求对象的起始地址必须是8字节的整数倍，也就是对象的大小必须是8字节的整数倍.因为对象头正好是8字节的1倍或者2倍，所以当实例数据没有对齐时，使用对齐填充来对齐。

# 38 对象的访问

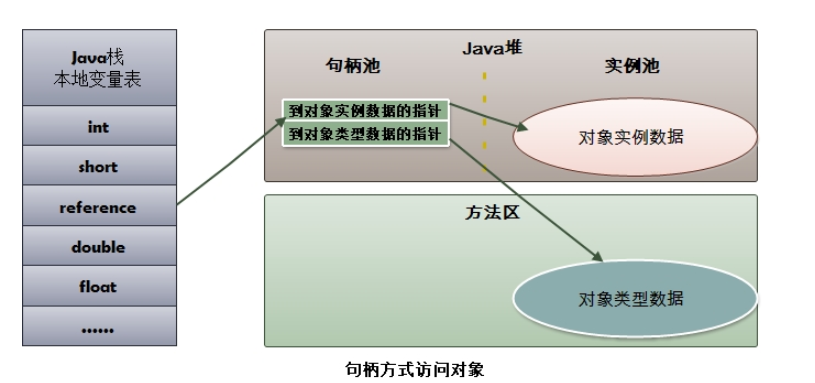
对象的访问是指通过栈上的reference数据来操作堆上的具体对象。

对象的访问主流的包括句柄和直接指针两种。

## 38.1 句柄访问

句柄方式会在堆中划分一块内存作为句柄池。句柄中包括2个指针，一个指向对象的实例数据，一个指向对象的类型数据。reference中存储的就是对象的句柄地址。

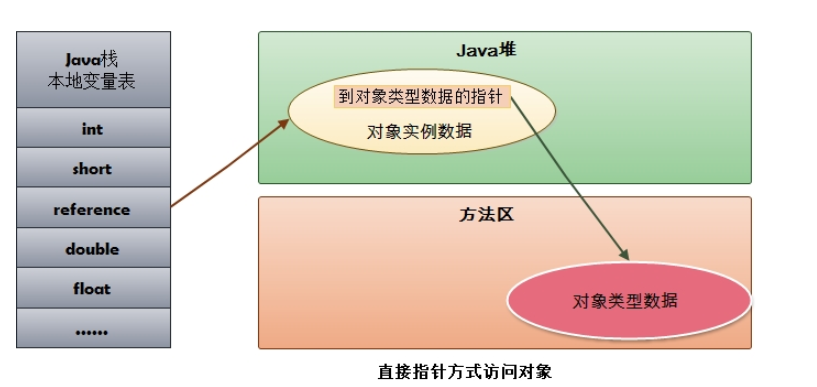
图1



## 38.2 直接指针访问

reference中存储的是对象的地址，因为对象的对象头中有一个指针指向了方法区的对象类型数据。

图1



## 38.3 句柄和直接指针的区别

1.句柄访问方式当对象在内存中移动时不需要改变reference的值，只需要改变句柄的值；而直接指针方式当对象在内存中移动时需要改变reference的值。

2.直接指针访问速度快，减少了一次定位对象实例数据的时间。

# 39 Java内存参数设置

## 39.1 -Xms和-Xmx和-Xmn 堆

-Xms：堆的最小值。

-Xmx：堆的最大值。

-Xmn：堆中的新生代的大小。

## 39.2 -XX:+HeapDumpOnOutOfMemoryError

-XX:+HeapDumpOnOutOfMemoryError：当出现内存溢出异常时，dump出当前的堆的转存快照，以便事后进行分析。

## 39.3 -Xoss和-Xss 栈

-Xoss：设置本地方法栈大小。

-Xss：本地方法栈+虚拟机栈大小。

（注意：HotSpot虚拟机中不区分本地方法栈和虚拟机栈，所以不需要用-Xoss设置，只需要-Xss设置栈大小。）

## 39.4 PermSize和MaxPermSize 方法区

PermSize：方法区的最小大小。

MaxPermSize：方法区的最大大小。

## 39.5 -XX:MaxDirectMemorySize

-XX:MaxDirectMemorySize：直接内存的最大值，如果不指定，则默认和堆的最大值-Xmx一样。

## 39.6 -XX:SurvivorRatio

-XX:SurvivorRatio=8表示新生代中Eden区和一个Survivor区的大小比值为8:1。

## 39.7 -XX:MaxTenuringThreshold

-XX:MaxTenuringThreshold表示新生代的年龄到达哪个阈值，就会被晋升到老年代。

## 39.8 -XX:PretenureSizeThreshold

-XX:PretenureSizeThreshold表示令大于这个阈值的对象直接在老年代分配。

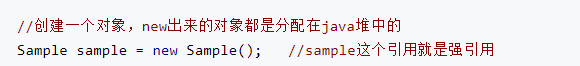
# 40 引用类型

原文链接：<https://blog.csdn.net/rodbate/article/details/72857447>

## 40.1 强引用

强引用就是我们平时使用的。垃圾回收器不会回收强引用的对象。

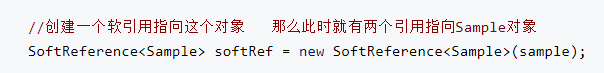
图1



## 40.2 软引用

软引用的对象垃圾回收器平时不会回收，但当将要发生内存溢出异常时，会回收软引用的对象。

图1



## 40.3 弱引用

弱引用的对象垃圾回收器无论内存是否足够都会回收。

图1



## 40.4 虚引用

虚引用的对象垃圾回收器无论内存是否足够都会回收。它的唯一作用是在对象被回收时能获取一个通知。

图1



# 41 注解

## 41.1 注解的分类

按注解的生命周期分：

1. 源码注解（RetentionPolocy.SOURCE）：表示该注解在编译时无效，所以该注解在class文中不存在；
2. 编译时注解（RetentionPolocy.CLASS）：表示注解在类加载时和类加载时有效，所以该注解在class文中存在，比如：@Override，@Deprecated，@SuppressWaring
3. 运行时注解（RetentionPolocy.RUNTIME）：表示该注解一直有效，即该注解在运行时会起作用，比如：@Autowired在运行时会注入bean。

按注解的来源分：

1. JDK注解
2. 第三方注解
3. 自定义注解

元注解：用来注解其他的注解。

## 41.2 元注解

元注解：用来注解其他的注解。

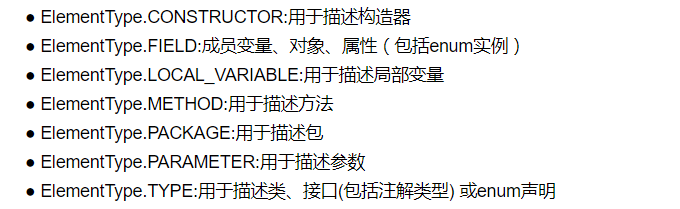
元注解分为以下4种：

@Documented：表示是否将该注解的信息添加到Java doc种。

@Retention：定义该注解的生命周期。

@Target：表示该注解可以用在什么地方.如图1。

图1



@Inherited：表示使用该注解的类的子类是否能继承该注解。

## 41.3 自定义注解

使用@interface定义一个注解，使用元注解@Documented，@Retention，@Target，@Inherited来修饰该注解。该注解中的方法其实是定义的成员变量，可以使用default来指定成员变量的默认值。

（注意：成员变量的类型是受限制的，只能是基本的数据类型，String，Class，Annotation，Enumeration等）

（注意：如果只有一个成员变量，成员名可以取value()，使用时可以忽略成员名和赋值号（=））

（注意：自定义注解类可以没有成员变量，这时候这个注解称为标识注解）

图1（定义自定义注解）

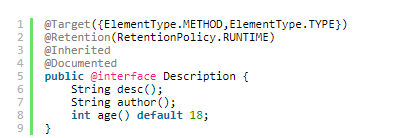
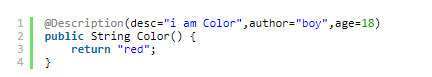


图2（使用该自定义注解）



## 41.4 Spring注解

@Import：导入一个普通的Java类，并将它声明为一个bean。

图1

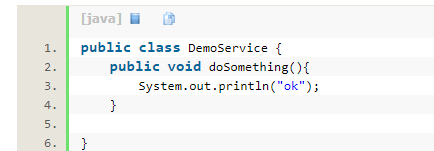
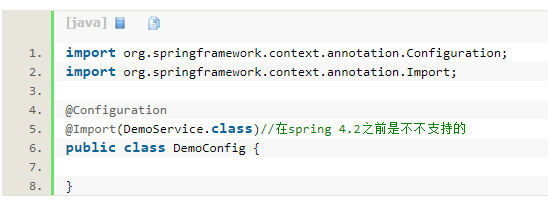


图2



# 42 JDK工具

## 42.1 jps

jps列出正在运行的虚拟机进程（VMID），以及该虚拟机进程的执行主类（main()函数所在的类），以及该虚拟机进程的本地虚拟机唯一ID（LVMID）。

图1（jps命令的参数）

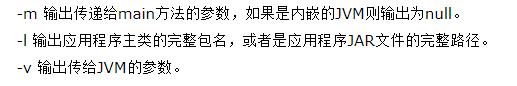
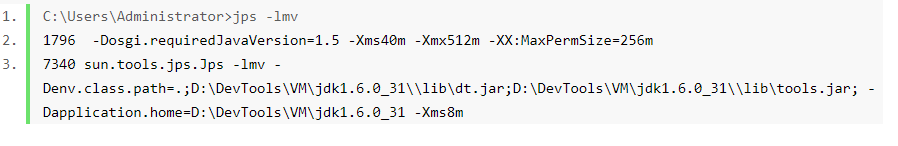


图2



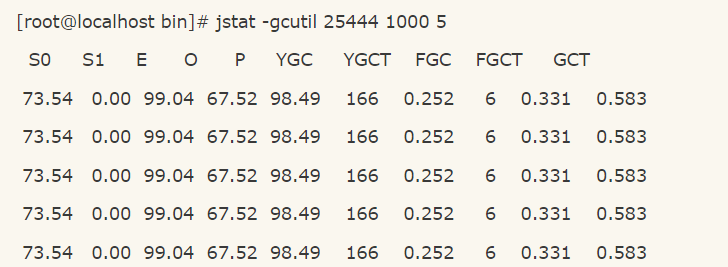
## 42.2 jstat

jstat显示本地或远程虚拟机进程中的类装载，内存，垃圾收集，JIT编译等运行数据。

-gc：查看Java堆（包括Eden区，两个Survivor区，老年代），永久代等的容量，已使用空间，GC合计时间等。

-gcutil ：和-gc基本一致，但更关注各个区的占用百分比。

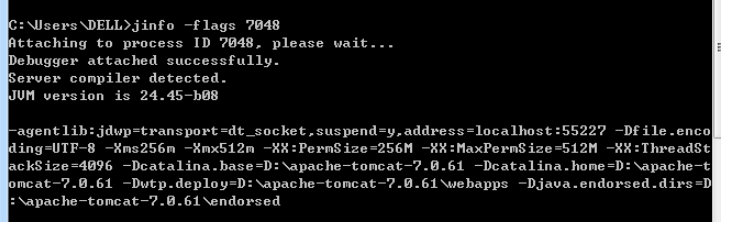
图1（jstat -gcutil 2764）



## 42.3 jinfo

jinfo查看和调整虚拟机各个参数的值。

图1



## 42.4 jmap

jmap生成堆转存储快照。

-histo：显示堆中对象占用情况，从高到低显示。

图1

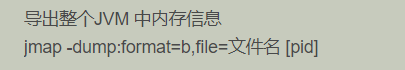
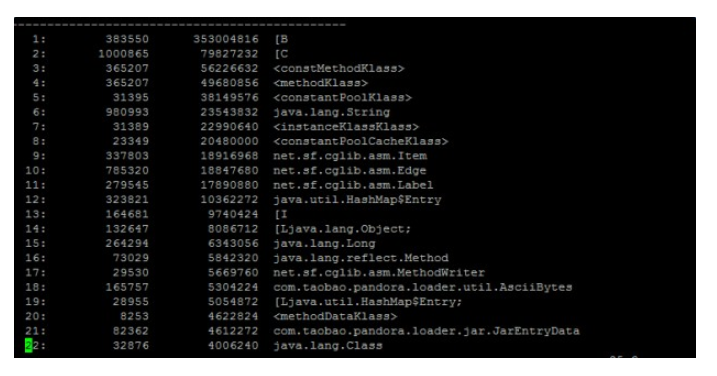


图2（jmap -histo:live pid，byte数组占用了3G）



## 42.5 jhat

jhat分析堆转存储快照。

## 42.6 jstack

jstack生成须奴籍当前的线程快照。

## 42.7 hsdis

hsdis将本地代码还原成汇编代码输出，并生成大量有价值的注释。

## 42.8 JConsole

JConsole是图形化的监控和管理控制台。

# 43 排序算法

原文链接：

<https://www.cnblogs.com/10158wsj/p/6782124.html?utm_source=tuicool&utm_medium=referral>

<https://blog.csdn.net/happy_wu/article/details/51841244>

排序算法分为内部排序和外部排序，内部排序是数据记录在内存中进行排序

图1

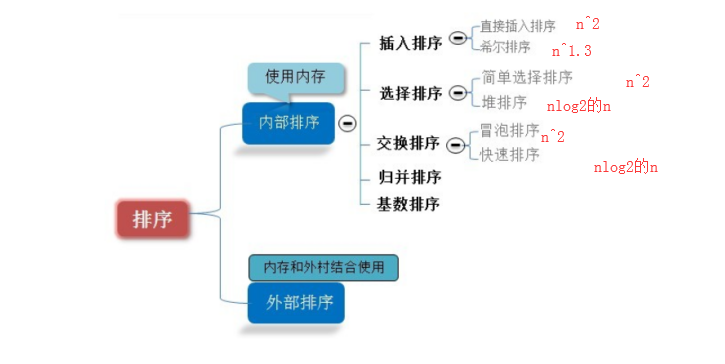


图2（时间复杂度）



## 43.1 直接插入排序

原理：从第二个元素开始，假设前面都是有序的，遍历它前面的每一个元素，找到一个比它小的元素然后插在这个元素后面。

图1（原理）

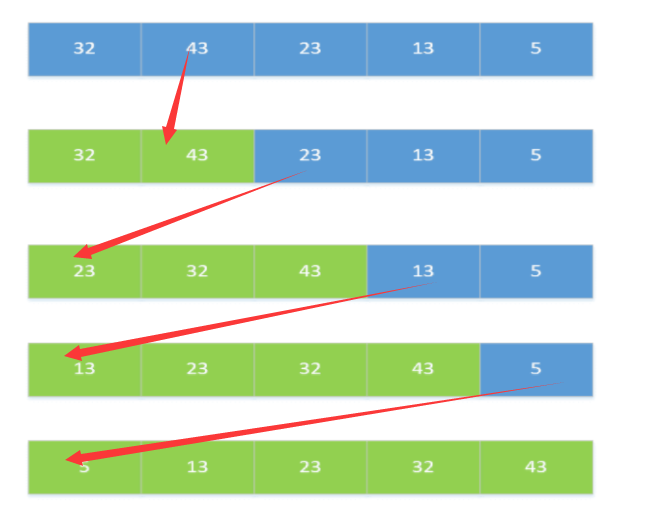
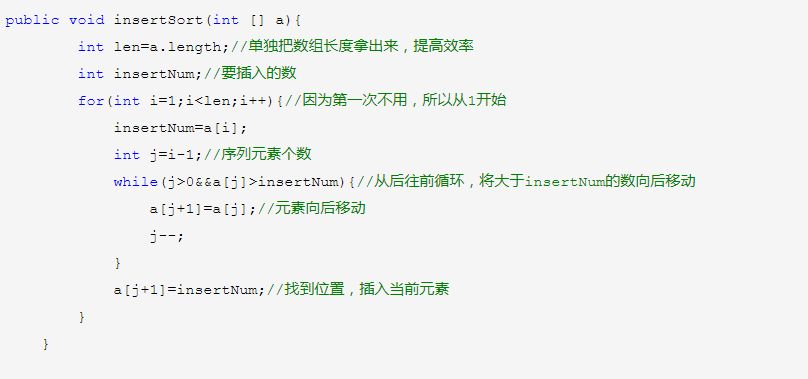


图2（代码实现）



## 43.2 希尔排序

希尔排序是针对直接插入排序的缺点改进的：

1. 直接排序在几乎已经排好序的数据中操作时效率高，但一般情况下是低效的。

原理：先将整个待排序的序列分割成若干个子序列分别进行直接插入排序，待整个序列“基本有序”时，再对全体序列进行直接插入排序。

图1（分组进行直接插入排序，当分组数为1时，即对整个序列进行直接插入排序）



## 43.3 简单选择排序

原理：从整个序列中选取最小（或最大）的数与第1个位置的数交换，然后在剩下的序列中选择最小（或最大）的数与第2个位置的数交换，以此类推。

图1（示例）

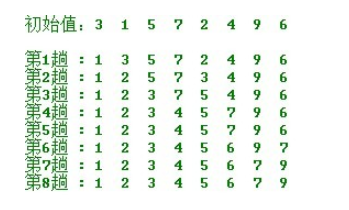


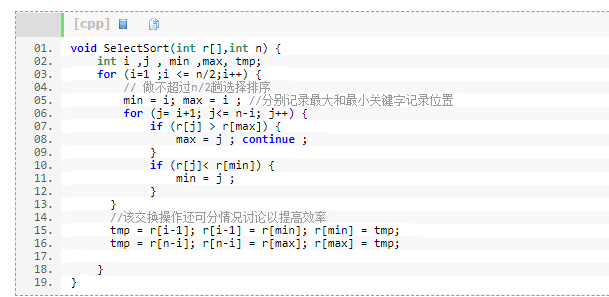
图2



## 43.4 二元选择排序

二元选择排序是对简单选择排序的改进，因为简单选择排序每次只能选择一个元素改变它的位置，改进后的二元选择排序每次可以选择一个最大和一个最小的元素改变它的位置。最多只需要进行n/2趟排序。

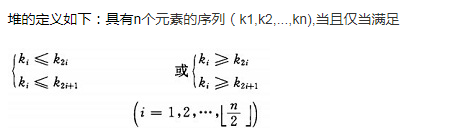
图1



## 43.5 堆排序

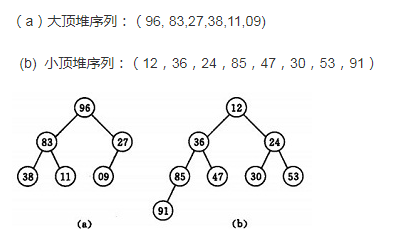
堆的定义：

图1



（注意：由堆的定义可知，堆顶元素（即序列的第一个元素）一定是最小项或者最大项）

图2（堆的定义图形化）



堆排序原理：将n个数的序列调整顺序成为一个堆，然后输出堆顶元素得到最小（或最大）值；然后将剩下n-1个数的序列重新调整顺序成为一个堆，再输出堆顶元素，以此类推。

调整最小堆的方法（如图3）：

1. 将堆顶元素（12）和最后一个元素（91）交换，此时堆被破坏。
2. 将根节点和左右子树中的较小者交换。
3. 如果和左子树交换后堆被破坏，则重复步骤2；如果和右子树交换后堆被破坏，则重复步骤2。

图3（调整最小堆的方法图解）

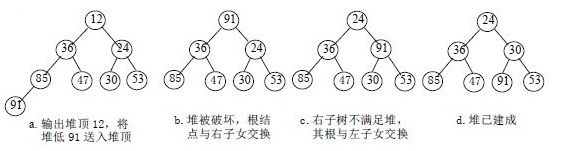
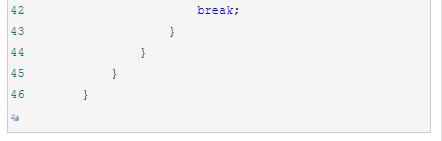


图4（代码实现）





## 43.6 快速排序

快速排序原理：选取一个基准元素，通常选第一个元素或最后一个元素，将比基准元素小的放在基准元素左边，将比基准元素大的放在基准元素右边。递归的对基准元素的左边和右边进行这样的操作。

图1（示例）

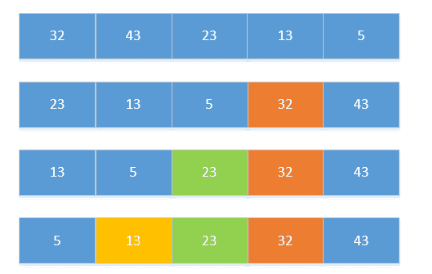


图2（代码实现）



## 43.7 冒泡排序

冒泡排序原理：将序列中的元素两两比较，将最大的放在最后面；在除最后一个元素以外的剩余元素两两比较，将最大的又放到最后面，以此类推。

图1（示例）

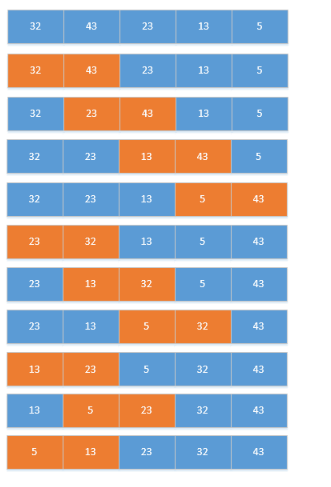
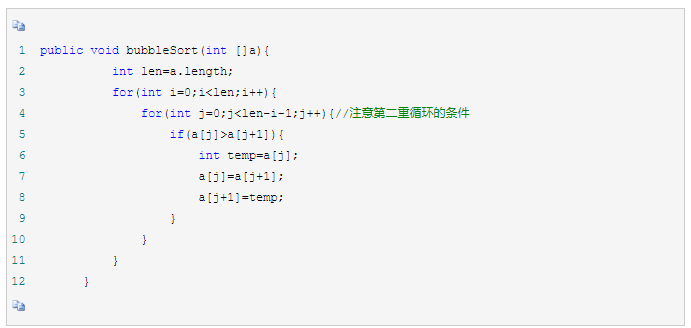


图2（代码实现）



# 44 二叉树

二叉树：每个节点最多有2个子节点，左子节点的关键值必须小于根节点的关键值，右子节点的关键值必须大于或等于根节点的关键值。

## 44.1 二叉树和数组，链表的区别

1.二叉树可以像数组一样快速的查找，也可以像链表一样快速的添加

2.但二叉树的删除操作复杂

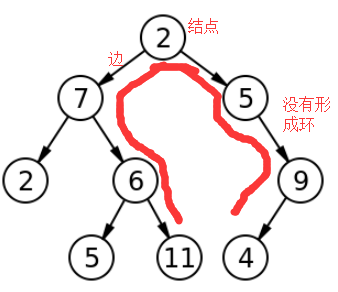
## 44.2 树的定义

原文链接：

<https://blog.csdn.net/HaoDaWang/article/details/78065162?locationNum=2&fps=1>

树是由结点和边组成的，且不存在任何环（即在图上找不到任何圈圈）的一种数据结构。

图1



树的术语：

度：一个结点拥有的子树个数。比如：二叉树的度最大为2。

结点的高度：一个结点到某个叶子结点之间的最长路径上的边的个数。树的高度就是根结点的高度。

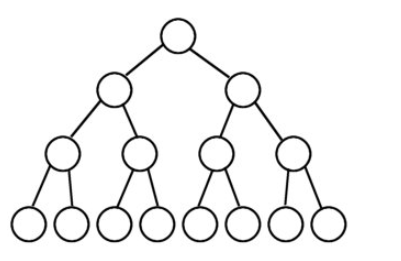
结点的深度：从树的根结点到该结点的边的个树。

边：一个结点和另一个结点之间的连接称为边。

## 44.3 完美二叉树

一个深度为k且有2^(k+1)-1个结点的二叉树称为完美二叉树，也叫满二叉树。

图1（深度为3，结点数=2^4-1=15）



## 44.4 完全二叉树

完全二叉树的倒数第二层往上都满足完美二叉树，最后一层可以不完全填充，但叶子结点都靠左对齐。

图1

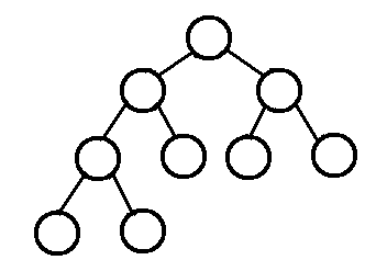
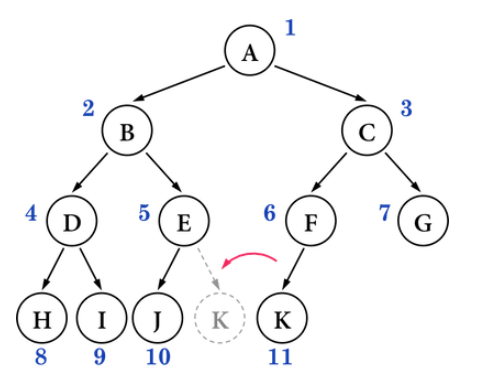


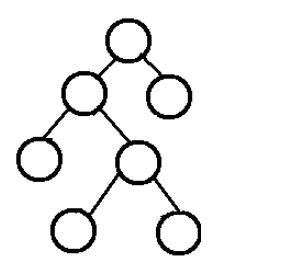
图2（注意这个不是完全二叉树，因为叶子结点没有靠左对齐）



## 44.5 完满二叉树

只有有孩子，必须是2个孩子。

图2



# 45 反射

反射是指，在运行过程中，对于任意一个类都可以获取它的所有属性和方法；对于任意一个对象都可以调用它的任意一个方法。

优点：

1. 不需要重新编译源码，因为反射是在运行时获取的。
2. 灵活，可以获取某个类的所有属性和方法进行某些操作。

缺点：

1. 反射的性能低。
2. 破坏了类的封装性，导致类私有的属性和方法都可以被反射获取到。