# 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个对象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）

# 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Colleaction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，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不直不能释放。

（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



# 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试一个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 ？？？

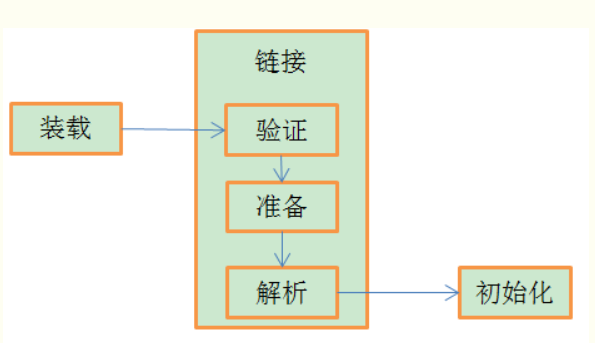
原文链接：<https://blog.csdn.net/justloveyou_/article/details/71713781>

# 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。

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）

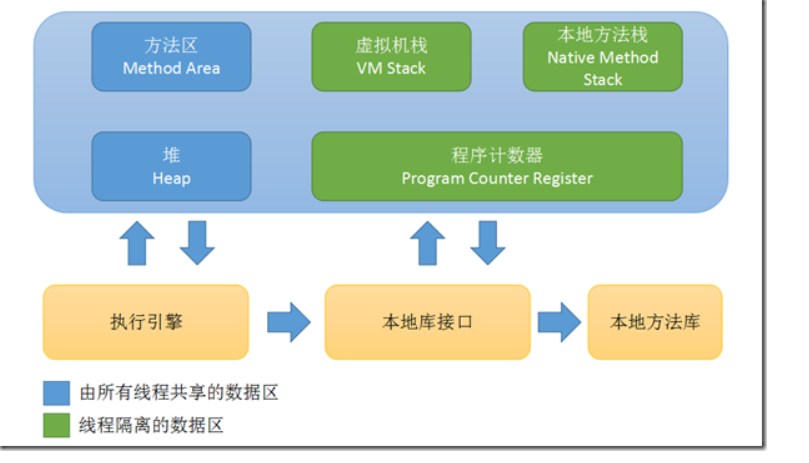
（注意：如果public static int value=123，在准备阶段过后，value的初始值是0，不是123，因为这时候还未执行任何Java方法；但如果是final修饰的，那准备阶段过后，value的初始值是123）

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 StackOverflowError

### 34.2.3 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 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。

## 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，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 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。

### 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的DirectByteBuf对象就在这个区域。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。