抽象类和接口异同：

相同：

都不能被直接实例化，都可以通过继承实现其抽象方法。  
都是面向抽象编程的技术基础，实现了诸多的设计模式

不同：

接口可以有多个，但只能继承一个

抽象类中可以定义非抽象方法，但接口中只有抽象方法

接口中基本数据类型为static 而抽类象不是的。

抽象类可以有构造器，接口不能有构造器

抽象方法可以有public、protected和default这些修饰符 接口方法默认修饰符是public。你不可以使用其它修饰符

抽象方法可以有main方法并且我们可以运行它，接口没有main方法，因此我们不能运行它。

抽象方法比接口速度要快，接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。

如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。如果你往接口中添加方法，那么你必须改变实现该接口的类。

内部类作用和分类：

http://www.cnblogs.com/chenssy/p/3388487.html

1. 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。
2. 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。
3. 创建内部类对象的时刻并不依赖于外围类对象的创建。
4. 内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。
5. 内部类提供了更好的封装，除了该外围类，其他类都不能访问

成员内部类：（不可以拥有static变量和方法）

public class OuterClass {

private String str;

public void outerDisplay(){

System.out.println("outerClass...");

}

public class InnerClass{

public void innerDisplay(){

//使用外围内的属性

str = "chenssy...";

System.out.println(str);

//使用外围内的方法

outerDisplay();

}

}

/\*推荐使用getxxx()来获取成员内部类，尤其是该内部类的构造函数无参数时 \*/

public InnerClass getInnerClass(){

return new InnerClass();

}

public static void main(String[] args) {

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.getInnerClass();

inner.innerDisplay();

}

}

--------

局部内部类（嵌套在方法内或作用域内，不常用）

public class Parcel5 {

public Destionation destionation(String str){

class PDestionation implements Destionation{

private String label;

private PDestionation(String whereTo){

label = whereTo;

}

public String readLabel(){

return label;

}

}

return new PDestionation(str);

}

public static void main(String[] args) {

Parcel5 parcel5 = new Parcel5();

Destionation d = parcel5.destionation("chenssy");

}

}

public class Parcel6 {

private void internalTracking(boolean b){

if(b){

class TrackingSlip{

private String id;

TrackingSlip(String s) {

id = s;

}

String getSlip(){

return id;

}

}

TrackingSlip ts = new TrackingSlip("chenssy");

String string = ts.getSlip();

}

}

public void track(){

internalTracking(true);

}

public static void main(String[] args) {

Parcel6 parcel6 = new Parcel6();

parcel6.track();

}

}

匿名内部类（通常是一个接口，直接当参数来传进去，需要现场实现）

静态内部类（嵌套内部类）：没有对外部的引用，而非静态内部类都有，但是这也要求它不能使用外围类的非static方法和变量

Is-a与has-a：

IS--A基于类继承或接口实现，IS-A是表达这句话的一种方式：“这个东西是那个东西的一种”。例如：野马是一种马。（是一个）  
HAS-A关系是基于用法（即引用）而不是继承。换句话说，如果类A中的代码具有对类B实例的引用，则“类A HAS-A类B”。例如：马有缰绳。（有一个）

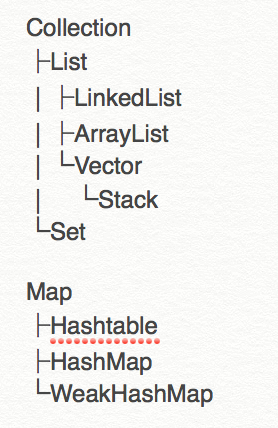
父类的静态方法能否被子类重写

1. **public** **class** Fu {
2. **public** **static** **void** show() {
3. System.out.println("父类的静态方法");
4. }
5. **public** **void** method() {
6. System.out.println("父类的一般方法");
7. }
8. }
9. **public** **class** Zi **extends** Fu {
10. **public** **static** **void** main(String[] args) {
11. Fu fu = **new** Zi();
12. fu.show();
13. fu.method();
14. }
15. **public** **static** **void** show() {
16. System.out.println("子类的静态");
17. }
18. **public** **void** method() {
19. System.out.println("子类的一般方法");
20. }
22. }

结果：

  父类的静态方法  
  子类的一般方法

Java集合继承关系

linkedHashmap继承hashmap实现了Map接口，还有ConcurrentHashmap继承自AbstractMap,实现了ConcurrentMap接口，AbstractMap是个抽象类，实现了Map接口

泛型中extends和super区别

<https://itimetraveler.github.io/2016/12/27/%E3%80%90Java%E3%80%91%E6%B3%9B%E5%9E%8B%E4%B8%AD%20extends%20%E5%92%8C%20super%20%E7%9A%84%E5%8C%BA%E5%88%AB%EF%BC%9F/>

两者其实是相反的

Plate<Fruit> p=new Plate<Apple>(new Apple());

这样的语句会报错（但实际上可以将尖括号中的Apple去掉，这里只是举例），需要写成

Plate<? extends Fruit> p=new Plate<Apple>(new Apple());

但是这两个关键字有副作用，extend不能往里存，只能取，而且取到的只能放在Fruit或者Fruit的基类里

Super存没问题，但是取只能放到Object里面（连fruit也不行）

Pesc原则

* **频繁往外读取内容的，适合用上界Extends。**
* **经常往里插入的，适合用下界Super。**

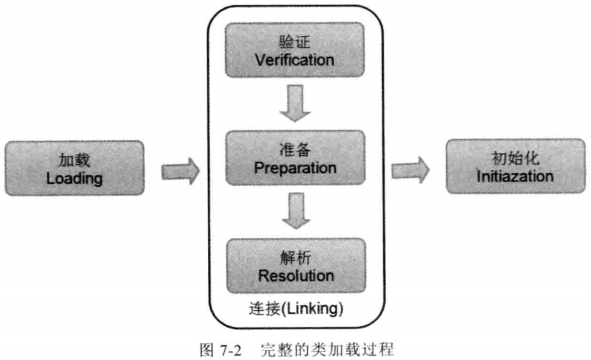
Java虚拟机特性

不同平台不需要重新编译，java虚拟机直接执行目标代码（字节码），解释成具体平台的机器指令即可

双亲委派模型

<https://segmentfault.com/a/1190000008995781>

类加载过程



* **加载**：加载阶段由类加载器负责，过程见类加载器;
* **验证**：验证阶段负责验证类数据信息是否符合JVM规范，是否是一个有效的字节码文件；
* **准备**：准备阶段负责为类中static变量分配空间，并初始化（与程序无关，系统初始化）；
* **解析**：解析阶段负责将常量池中所有符号引用转换为直接引用；
* **初始化**：初始化阶段负责将所有static域按照程序指定操作对应执行（赋值static变量，执行static块）

上述阶段通常都是交叉混合允许，没有严格的先后执行顺序；

类加载器的主要任务：是**类加载过程中的加载操作**：**根据一个类的全限定名读取该类的二进制字节流到JVM内部，然后转换为一个对应的java.lang.Class对象实例**；  
开发者可以通过编写自定义类加载器来自定义类的加载规则

类加载器分类

启动类加载器/Bootstrap ClassLoader

在HotSpot虚拟机中，Bootstrap ClassLoader用C++语言编写并嵌入JVM内部，主要负载加载JAVA\_HOME/lib目录中的所有类，或者加载由选项-Xbootcalsspath指定的路径下的类；

拓展类加载器/ExtClasLoader

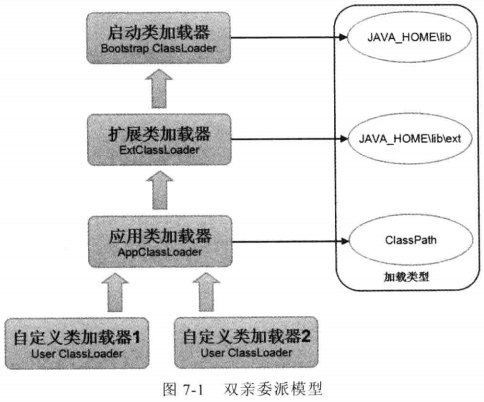
ExtClassLoader继承ClassLoader类，负载加载JAVA\_HOME/lib/ext目录中的所有类型，或者由参数-Xbootclasspath指定路径中的所有类型；

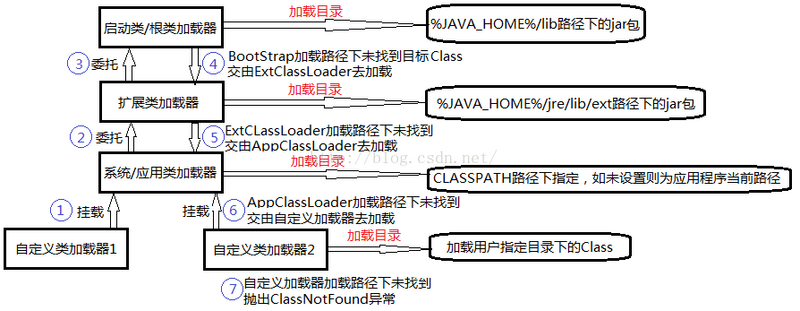
应用程序类加载器/AppClassLoader

ExtClassLoader继承ClassLoader类，负责加载用户类路径ClassPath下的所有类型，一般情况下为程序的默认类加载器；

自定义加载器

Java虚拟机规范将所有继承抽象类java.lang.ClassLoader的类加载器，定义为**自定义类加载器**；





**双亲委派过程**：当一个类加载器收到类加载任务时，立即将任务委派给它的父类加载器去执行，直至委派给最顶层的启动类加载器为止。如果父类加载器无法加载委派给它的类时，将类加载任务退回给它的下一级加载器去执行;  
除了启动类加载器以外，每个类加载器拥有一个父类加载器，用户的自定义类加载器的父类加载器是AppClassLoader；  
双亲委派模型可以保证全限名指定的类，只被加载一次；  
双亲委派模型不具有强制性约束，是Java设计者推荐的类加载器实现方式；

Java垃圾回收：

Java 垃圾回收机制最基本的做法是分代回收。

Young Generation Space

Tenure Generation Space

Permanent Space存储不变的类定义、字节码和常量等

几种常用算法：

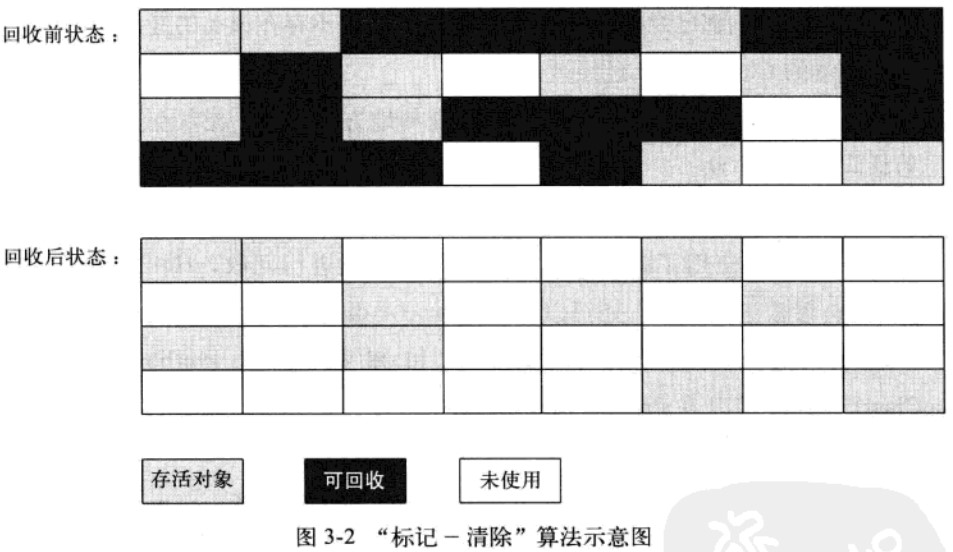
标记-清除

分为标记和清除两个阶段。首先标记出所有需要回收的对象，在标记完成后统一回收被标记的对象

有什么缺点？

1. 效率问题。标记和清除过程的效率都不高。

2. 空间问题。标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能导致，程序分配较大对象时无法找到足够的连续内存，不得不提前出发另一次垃圾收集动作。



**复制算法（Copying）- 新生代**

将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当这一块的内存用完了，就将存活着的对象复制到另一块上面，然后再把已经使用过的内存空间一次清理掉。

主要用来存储新创建的对象，内存较小，垃圾回收频繁。这个区又分为三个区域：一个 Eden Space 和两个 Survivor Space。

当对象在堆创建时，将进入年轻代的Eden Space。

垃圾回收器进行垃圾回收时，扫描Eden Space和A Suvivor Space，如果对象仍然存活，则复制到B Suvivor Space，如果B Suvivor Space已经满，则复制 Old Gen

扫描A Suvivor Space时，如果对象已经经过了几次的扫描仍然存活，JVM认为其为一个Old对象，则将其移到Old Gen。

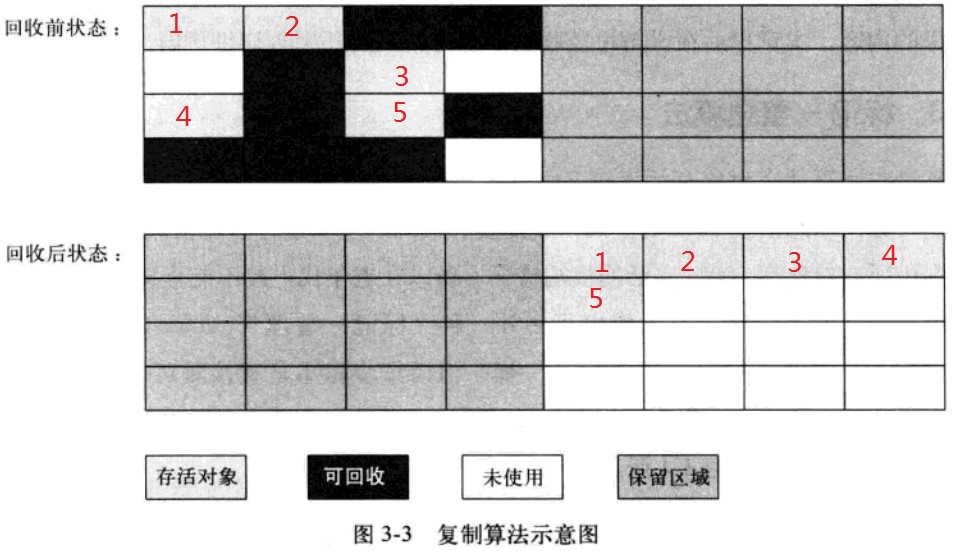
扫描完毕后，JVM将Eden Space和A Suvivor Space清空，然后交换A和B的角色（即下次垃圾回收时会扫描Eden Space和B Suvivor Space。

优点？

复制算法使得每次都是针对其中的一块进行内存回收，内存分配时也不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

缺点？

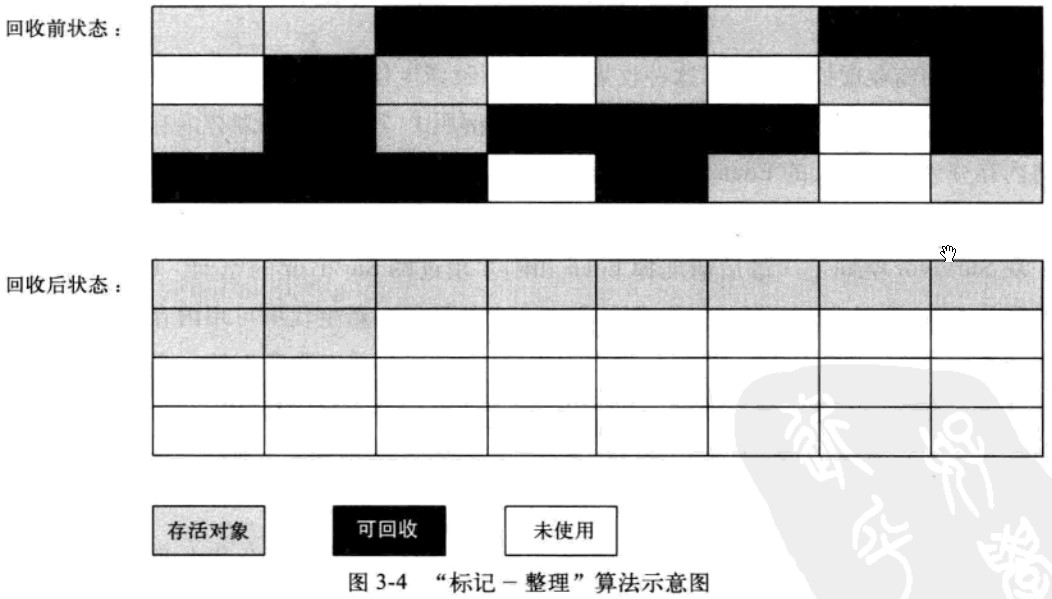
将内存缩小为原来的一半。在对象存活率较高时，需要执行较多的复制操作，效率会变低。



标记-整理算法（Mark-Compact）-老年代

标记过程仍然与“标记-清除”算法一样，但不是直接对可回收对象进行清理，而是让所有存活的对象向一端移动，然后直接清理掉边界以外的内存。

主要用来存储长时间被引用的对象。它里面存放的是经过几次在 Young Generation Space 进行扫描判断过仍存活的对象，内存较大，垃圾回收频率较小。



分代收集算法：

根据对象的存活周期，将内存划分为几块。一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点，采用最适当的收集算法。

新生代：每次垃圾收集时会有大批对象死去，只有少量存活，所以选择复制算法，只需要少量存活对象的复制成本就可以完成收集。

老年代：对象存活率高、没有额外空间对它进行分配担保，必须使用“标记-清理”或“标记-整理”算法进行回收。

== equals和hashcode()区别

基本数据类型==是对值的检测，但是如果是复合数据类型，那么==是对地址的检查

Equals默认检查的是地址（Object类中）

Hashcode（）其实和他们没有太大关系，只是某些数据结构加速用到的工具，但是必须符合一些约定，比如equal返回相同的值，那么hashcode理应也要返回相同的值

Hashmap实现原理

1. HashMap概述： HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
2. HashMap的数据结构： 在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

但在java8之后，链表改为了树结构

**Java多态**

什么是多态

面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实 际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。 下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

多态存在的三个必要条件 一、要有继承； 二、要有重写； 三、父类引用指向子类对象。

多态的好处：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。

2.可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图8.3 所示。图中超类Shape规定了两个实现多态的接口方法，computeArea()以及computeVolume()。子类，如Circle和Sphere为了实现多态，完善或者覆盖这两个接口方法。

4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。

5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

Java中多态的实现方式：接口实现，继承父类进行方法重写，同一个类中进行方法重载。

**线程阻塞**

[**http://www.jackywang.tech/AndroidInterview-Q-A/chinese/java/%E4%BB%80%E4%B9%88%E5%AF%BC%E8%87%B4%E7%BA%BF%E7%A8%8B%E9%98%BB%E5%A1%9E-58-%E7%BE%8E%E5%9B%A2.html**](http://www.jackywang.tech/AndroidInterview-Q-A/chinese/java/%E4%BB%80%E4%B9%88%E5%AF%BC%E8%87%B4%E7%BA%BF%E7%A8%8B%E9%98%BB%E5%A1%9E-58-%E7%BE%8E%E5%9B%A2.html)

动机：

解决对共享存储区的访问冲突，同步不够用

方法：

Sleep

Suspend和Resume

Yield：换到下一个进程

Wait和notify：和前面几个区别是阻塞时会释放锁，属于Thread类，而前面几个都是Object类

另外前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放

另外：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

**并发编程**

**Todo**

**Hashmap和hashtable区别**

**1、 继承和实现区别**

　　Hashtable是基于陈旧的Dictionary类，完成了Map接口；HashMap是Java 1.2引进的Map接口的一个实现（HashMap继承于AbstractMap,AbstractMap完成了Map接口）。

**2、 线程安全不同**

　　HashTable的方法是同步的，HashMap是未同步，所以在多线程场合要手动同步HashMap。

**3、 对null的处理不同**

　　HashTable不允许null值(key和value都不可以),HashMap允许null值(key和value都可以)。即 HashTable不允许null值其实在编译期不会有任何的不一样，会照样执行，只是在运行期的时候Hashtable中设置的话回出现空指针异常。 HashMap允许null值是指可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示 HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。

**4、 方法不同**

　　HashTable有一个contains(Object value)，功能和containsValue(Object value)功能一样。

**5、HashTable使用Enumeration，HashMap使用Iterator。**

**6、HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。**

**7、哈希值的使用不同，HashTable直接使用对象的hashCode，**HashMap重新计算hash值，而且用与代替求模

**Hashmap和ArrayMap**

http://lvable.com/?p=217

Java库里的HashMap其实是一个连续的链表数组，通过让key计算hash值后插入对应的index里。当hash值发生碰撞时，可以采用线性探测，二次hash，或者后面直接变成链表的结构来避免碰撞。因为hash的值不是连续的，**所以hashmap实际需要占用的大小会比它实际能装的item的容量要大**

当HashMap记录存入的item size大于threshold后，HashMap就会进行扩容（resize）。当我们第一次新建一个HashMap对象的时候，默认的容量是16，若你只打算在HashMap里放入3个元素那将浪费至少13个空间。

他用两个数组来模拟Map，第一个数组存放存放item的hash值，第二数组是把key，value连续的存放在数组里，通过先算hash在第一个数组里找到它的hash index，根据这个index在去第二个数组里找到这个key-value。

在这里，在第一个数组里查找hash index的方法当然是用二分查找啦

这个数据结构的设计就做到了，有多个item我就分配多少内存，做到了memory的节约。并且因为数据结构是通过数组组织的，所以遍历的时候可以用index直接遍历也是很方便的有没有！但是缺点也很明显，查找达不到HashMap O(1)的查找时间。

当要存储的对象较少的时候（1000以下的时候）可以考虑用ArrayMap来减少内存的占用。

**其他区别**

//todo