# [Interface与abstract类的区别](https://www.cnblogs.com/mengchunchen/p/7890907.html)

## 定义

含有abstract修饰符的class即为抽象类，abstract 类不能创建的实例对象。含有abstract方法的类必须定义为abstract class，abstract class类中的方法不必是抽象的。abstract class类中定义抽象方法必须在具体(Concrete)子类中实现，所以，不能有抽象构造方法或抽象静态方法。如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract类型。

接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象的。接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为public static final。

## 区别

### 语法上的区别

1.抽象类可以有构造方法，接口中不能有构造方法。

2.抽象类中可以有普通成员变量，接口中没有普通成员变量

3.抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。

4. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型（1.9版本上接口可以有私有方法）。

5. 抽象类中可以包含静态方法，接口中不能包含静态方法

6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。

7. 一个类可以实现多个接口，但只能继承一个抽象类。

### 应用上的区别

接口更多的是在系统架构设计方法发挥作用，主要用于定义模块之间的通信契约。而抽象类在代码实现方面发挥作用，可以实现代码的重用，例如，模板方法设计模式是抽象类的一个典型应用，假设某个项目的所有Servlet类都要用相同的方式进行权限判断、记录访问日志和处理异常，那么就可以定义一个抽象的基类，让所有的Servlet都继承这个抽象基类，在抽象基类的service方法中完成权限判断、记录访问日志和处理异常的代码，在各个子类中只是完成各自的业务逻辑代码。

# IO 和 NIO的区别，NIO优点

## 定义

从硬盘读取数据，然后程序一直等，数据读完后，继续操作。这种方式是最简单的，叫**阻塞IO**

从硬盘读取数据，然后程序继续向下执行，等数据读取完后，通知当前程序（对硬件来说叫**中断**，对程序来说叫**回调**），然后此程序可以立即处理数据，也可以执行完当前操作在读取数据，这个叫做非阻塞IO，也就是NIO

## 区别

### java.NIO包里包括三个基本的组件

buffer：因为NIO是基于缓冲的，所以buffer是最底层的必要类，这也是IO和NIO的根本不同，虽然stream等有buffer开头的扩展类，但只是流的包装类，还是从流读到缓冲区，而NIO却是直接读到buffer中进行操作。

channel：类似于IO的stream，但是不同的是除了FileChannel，其他的channel都能以非阻塞状态运行。FileChannel执行的是文件的操作，可以直接DMA操作内存而不依赖于CPU。其他比如socketchannel就可以在数据准备好时才进行调用。

Selector（选择器）：用于分发请求到不同的channel，这样才能确保channel不处于阻塞状态就可以收发消息。Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### 面向流和面向缓冲

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

### 阻塞与非阻塞

Java IO流都是阻塞的，这意味着，当一条线程执行read()或者write()方法时，这条线程会一直阻塞直到读取到了一些数据或者要写出去的数据已经全部写出，在这期间这条线程不能做任何其他的事情。

java NIO的非阻塞模式(Java NIO有阻塞模式和非阻塞模式，阻塞模式的NIO除了使用Buffer存储数据外和IO基本没有区别)允许一条线程从channel中读取数据，通过返回值来判断buffer中是否有数据，如果没有数据，NIO不会阻塞，因为不阻塞这条线程就可以去做其他的事情，过一段时间再回来判断一下有没有数据。NIO的写也是一样的，一条线程将buffer中的数据写入channel，它不会等待数据全部写完才会返回，而是调用完write()方法就会继续向下执行。

## 优势

在老的IO包中，serverSocket和socket都是阻塞式的，因此一旦有大规模的并发行为，而每一个访问都会开启一个新线程。这时会有大规模的线程上下文切换操作（因为都在等待，所以资源全都被已有的线程吃掉了），这时无论是等待的线程还是正在处理的线程，响应率都会下降，并且会影响新的线程。

而NIO包中的serverSocket和socket就不是这样，只要注册到一个selector中，当有数据放入通道的时候，selector就会得知哪些channel就绪，这时就可以做响应的处理，这样服务端只有一个线程就可以处理大部分情况（当然有些持续性操作，比如上传下载一个大文件，用NIO的方式不会比IO好）。

# Jdk1.7和1.8新特性

## Jdk1.7新特性

1、switch中可以使用字符串了

2、运用List<String> tempList = new ArrayList<>();即泛型实例化类型自动推断

3、

## Jdk1.8新特性

1、Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可

2、新日期API

**Clock对象**

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

Clock clock = Clock.systemDefaultZone();

long millis = clock.millis();

Instant instant = clock.instant();

Date legacyDate = Date.from(instant);

**Timezones 时区**

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。 时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

System.out.println(ZoneId.getAvailableZoneIds());

// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");

ZoneId zone2 = ZoneId.of("Brazil/East");

System.out.println(zone1.getRules());

System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]

// ZoneRules[currentStandardOffset=-03:00]

**LocalTime 本地时间**

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

LocalTime now1 = LocalTime.now(zone1);

LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);

long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3

System.out.println(minutesBetween); // -239

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串

LocalTime late = LocalTime.of(23, 59, 59);

System.out.println(late); // 23:59:59

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedTime(FormatStyle.SHORT)

.withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);

System.out.println(leetTime); // 13:37

**LocalDate 本地日期**

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

LocalDate today = LocalDate.now();

LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);

LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);

DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedDate(FormatStyle.MEDIUM)

.withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);

System.out.println(xmas); // 2014-12-24

LocalDateTime 本地日期时间

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();

System.out.println(dayOfWeek); // WEDNESDAY

Month month = sylvester.getMonth();

System.out.println(month); // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);

System.out.println(minuteOfDay); // 1439

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

DateTimeFormatter formatter =

DateTimeFormatter

.ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);

String string = formatter.format(parsed);

System.out.println(string); // Nov 03, 2014 - 07:13

3、Optional类

Optional实际上是个容器：它可以保存类型T的值，或者仅仅保存null

# 类加载器

”.java”文件经过Java编译器编译成拓展名为”.class”的文件，”.class”文件中保存着Java代码经转换后的虚拟机指令，当需要使用某个类时，虚拟机将会加载它的”.class”文件，并创建对应的class对象，将class文件加载到虚拟机的内存，这个过程称为类加载

## 类加载的过程

加载：类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象

验证：目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。

准备：为类变量(即static修饰的字段变量)分配内存并且设置该类变量的初始值即0(如static int i=5;这里只将i初始化为0，至于5的值将在初始化时赋值)，这里不包含用final修饰的static，因为final在编译的时候就会分配了，注意这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

解析：主要将常量池中的符号引用替换为直接引用的过程。符号引用就是一组符号来描述目标，可以是任何字面量，而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，接口方法解析(这里涉及到字节码变量的引用，如需更详细了解，可参考《深入Java虚拟机》)。

初始化：类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量(如前面只初始化了默认值的static变量将会在这个阶段赋值，成员变量也将被初始化)。

**1.加载：（重点）**  
加载阶段是“类加载机制”中的一个阶段，这个阶段通常也被称作“装载”，主要完成：  
1.通过“类全名”来获取定义此类的二进制字节流

2.将字节流所代表的静态存储结构转换为方法区的运行时[**数据结构**](http://lib.csdn.net/base/datastructure)

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

相对于类加载过程的其他阶段，加载阶段(准备地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，也可以由用户自定义的类加载器完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机并未规定此区域的具体数据结构。然后在java堆中实例化一个java.lang.Class类的对象，这个对象作为程序访问方法区中的这些类型数据的外部接口。

**2.验证：（了解）**

验证是链接阶段的第一步，这一步主要的目的是确保class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身安全。  
验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1.文件格式验证

 验证class文件格式规范，例如： class文件是否已魔术0xCAFEBABE开头 ， 主、次版本号是否在当前虚拟机处理范围之内等

2.元数据验证

这个阶段是对字节码描述的信息进行语义分析，以保证起描述的信息符合java语言规范要求。验证点可能包括：这个类是否有父类(除了java.lang.Object之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被final修饰的)、如果这个类的父类是抽象类，是否实现了起父类或接口中要求实现的所有方法。

3.字节码验证

 进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

4.符号引用验证

符号引用中通过字符串描述的全限定名是否能找到对应的类、符号引用类中的类，字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

**3.准备：（了解）**

准备阶段是正式为类变量分配内存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。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(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。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value  = 12;

那么变量value在准备阶段过后的初始值为0而不是12，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为12的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，建设上面类变量value定义为：

public static final int value = 123;

编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value设置为123。

**4.解析：（了解）**  
解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。  
符号引用：符号引用是一组符号来描述所引用的目标对象，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范并没有规定解析阶段发生的具体时间，只要求了在执行anewarry、checkcast、getfield、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield和putstatic这13个用于操作符号引用的字节码指令之前，先对它们使用的符号引用进行解析，所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析的动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。分别对应编译后常量池内的CONSTANT\_Class\_Info、CONSTANT\_Fieldref\_Info、CONSTANT\_Methodef\_Info、CONSTANT\_InterfaceMethoder\_Info四种常量类型。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

**5.初始化：（了解）**

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。在以下四种情况下初始化过程会被触发执行：

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、读取或设置一个类的静态字段(被final修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用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

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、则需要先出发其父类的初始化

4.jvm启动时，用户指定一个执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value  = 12;  在准备阶段完成后 value的值为0，而在初始化阶调用了类构造器<clinit>()方法，这个阶段完成后value的值为12。

\*类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句快可以赋值，但是不能访问。

\*类构造器<clinit>()方法与类的构造函数(实例构造函数<init>()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

\*由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句快要优先于子类的变量赋值操作。

\*<clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，那么编译器可以不为这个类生成<clinit>()方法。

\*接口中不能使用静态语句块，但接口与类不太能够的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

\*虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

## 类加载器类型

在虚拟机提供了3种类加载器，引导（Bootstrap）类加载器、扩展（Extension）类加载器、系统（System）类加载器（也称应用类加载器）

### 启动（Bootstrap）类加载器

启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，它负责将 <JAVA\_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中，注意必由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类)。

### 扩展（Extension）类加载器

扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher$ExtClassLoader类，由Java语言实现的，是Launcher的静态内部类，它负责加载<JAVA\_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定位路径中的类库，开发者可以直接使用标准扩展类加载器。

### 系统（System）类加载器

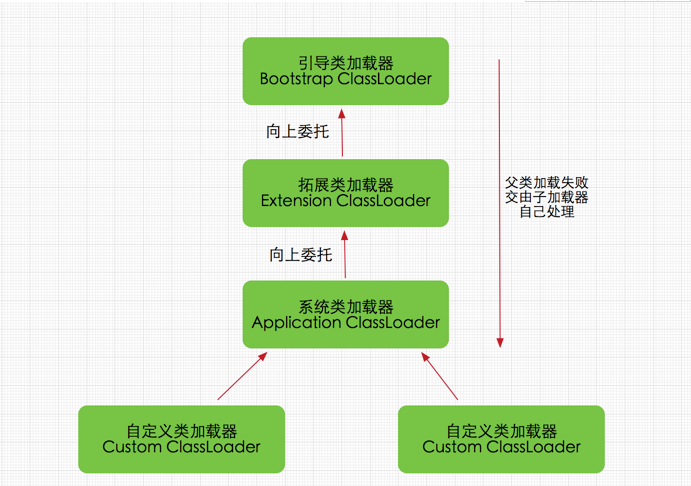
也称应用程序加载器是指 Sun公司实现的sun.misc.Launcher$AppClassLoader。它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认的类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，需要注意的是，Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类处理，它一种任务委派模式，下面我们进一步了解它。

### 自定义类加载器

## 双亲委派模式

双亲委派模式要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器，请注意双亲委派模式中的父子关系并非通常所说的类继承关系，而是采用组合关系来复用父类加载器的相关代码，类加载器间的关系如下：



双亲委派模式是在Java 1.2后引入的，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完成，这不就是传说中的实力坑爹啊？那么采用这种模式有啥用呢?

### 双亲委派模式的优势

采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出如下异常：

java.lang.SecurityException: Prohibited package name: java.lang

## 类加载器相关方法

loadClass(String)

该方法加载指定名称（包括包名）的二进制类型，该方法在JDK1.2之后不再建议用户重写但用户可以直接调用该方法，loadClass()方法是ClassLoader类自己实现的，该方法中的逻辑就是双亲委派模式的实现，其源码如下，loadClass(String name, boolean resolve)是一个重载方法，resolve参数代表是否生成class对象的同时进行解析相关操作。

findClass(String)

在JDK1.2之前，在自定义类加载时，总会去继承ClassLoader类并重写loadClass方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadClass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中，从前面的分析可知，findClass()方法是在loadClass()方法中被调用的，当loadClass()方法中父加载器加载失败后，则会调用自己的findClass()方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委托模式。需要注意的是ClassLoader类中并没有实现findClass()方法的具体代码逻辑，取而代之的是抛出ClassNotFoundException异常，同时应该知道的是findClass方法通常是和defineClass方法一起使用的(稍后会分析)，ClassLoader类中findClass()方法源码如下：

defineClass(byte[] b, int off, int len)

defineClass()方法是用来将byte字节流解析成JVM能够识别的Class对象(ClassLoader中已实现该方法逻辑)，通过这个方法不仅能够通过class文件实例化class对象，也可以通过其他方式实例化class对象，如通过网络接收一个类的字节码，然后转换为byte字节流创建对应的Class对象，defineClass()方法通常与findClass()方法一起使用，一般情况下，在自定义类加载器时，会直接覆盖ClassLoader的findClass()方法并编写加载规则，取得要加载类的字节码后转换成流，然后调用defineClass()方法生成类的Class对象。

resolveClass(Class≺?≻ c)

使用该方法可以使用类的Class对象创建完成也同时被解析。前面我们说链接阶段主要是对字节码进行验证，为类变量分配内存并设置初始值同时将字节码文件中的符号引用转换为直接引用。

上述4个方法是ClassLoader类中的比较重要的方法，也是我们可能会经常用到的方法。接看SercureClassLoader扩展了 ClassLoader，新增了几个与使用相关的代码源(对代码源的位置及其证书的验证)和权限定义类验证(主要指对class源码的访问权限)的方法，一般我们不会直接跟这个类打交道，更多是与它的子类URLClassLoader有所关联，前面说过，ClassLoader是一个抽象类，很多方法是空的没有实现，比如 findClass()、findResource()等。而URLClassLoader这个实现类为这些方法提供了具体的实现，并新增了URLClassPath类协助取得Class字节码流等功能，在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承URLClassLoader类，这样就可以避免自己去编写findClass()方法及其获取字节码流的方式，使自定义类加载器编写更加简洁，下面是URLClassLoader的类图(利用IDEA生成的类图)

## 为什么要自定义类加载器

我们需要的类不一定存放在已经设置好的classPath下(有系统类加载器AppClassLoader加载的路径)，对于自定义路径中的class类文件的加载，我们需要自己的ClassLoader

有时我们不一定是从类文件中读取类，可能是从网络的输入流中读取类，这就需要做一些加密和解密操作，这就需要自己实现加载类的逻辑，当然其他的特殊处理也同样适用。

可以定义类的实现机制，实现类的热部署,如OSGi中的bundle模块就是通过实现自己的ClassLoader实现的。

# 关键字

## final、finally、finalize

1、final修饰符（关键字）。被final修饰的类，就意味着不能再派生出新的子类，不能作为父类而被子类继承。因此一个类不能既被abstract声明，又被final声明。将变量或方法声明为final，可以保证他们在使用的过程中不被修改。被声明为final的变量必须在声明时给出变量的初始值，而在以后的引用中只能读取。被final声明的方法也同样只能使用，即不能方法重写。

2、finally是在异常处理时提供finally块来执行任何清除操作。不管有没有异常被抛出、捕获，finally块都会被执行。try块中的内容是在无异常时执行到结束。catch块中的内容，是在try块内容发生catch所声明的异常时，跳转到catch块中执行。finally块则是无论异常是否发生，都会执行finally块的内容，所以在代码逻辑中有需要无论发生什么都必须执行的代码，就可以放在finally块中。

3、finalize是方法名。java技术允许使用finalize（）方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在object类中定义的，因此所有的类都继承了它。子类覆盖finalize（）方法以整理系统资源或者被执行其他清理工作。finalize（）方法是在垃圾收集器删除对象之前对这个对象调用的。

## volatile

volatile可以保证java 共享变量的可见性和有序性。

### 保证可见性场景

一个线程池更改某个共享变量的值，一个线程读取某个共享变量的值，此时如果该变量未使用volatile，即使先改变了某个共享变量的值，另一个线程读取的值仍旧是改变前的值。

### 保证有序性场景

Doublecheck实现单例模式

先判断变量是否为空，为空则进行同步锁，再判断变量是否为空，如果为空则创建对象并赋值给改不了。

对象创建可以分为三个步骤：

（1）分配内存空间。

（2）初始化对象。

（3）将内存空间的地址赋值给对应的引用。

但是由于操作系统可以对指令进行重排序，所以上面的过程也可能会变成如下过程：

（1）分配内存空间。

（2）将内存空间的地址赋值给对应的引用。

（3）初始化对象

这样就会导致doublecheck失效，即使变量值不为空，但实际上变量所指向的内存地址中还未初始化对象，造成后续不可预料的错误。

而添加volatile的变量，则可以保证有序性，从而只有初始化对象后再将内存地址复制给引用变量，保证check的正确性。

### Volatile的原理

通过上面的例子，我们基本应该知道了volatile是什么以及怎么使用。现在我们再来看看volatile的底层是怎么实现的。

**1、可见性实现：**

在前文中已经提及过，线程本身并不直接与主内存进行数据的交互，而是通过线程的工作内存来完成相应的操作。这也是导致线程间数据不可见的本质原因。因此要实现volatile变量的可见性，直接从这方面入手即可。对volatile变量的写操作与普通变量的主要区别有两点：

（1）修改volatile变量时会强制将修改后的值刷新的主内存中。

（2）修改volatile变量后会导致其他线程工作内存中对应的变量值失效。因此，再读取该变量值的时候就需要重新从读取主内存中的值。

通过这两个操作，就可以解决volatile变量的可见性问题。

2、有序性实现：

 在解释这个问题前，我们先来了解一下Java中的happen-before规则，JSR 133中对Happen-before的定义如下：

通俗一点说就是如果a happen-before b，则a所做的任何操作对b是可见的。（这一点大家务必记住，因为happen-before这个词容易被误解为是时间的前后）。我们再来看看JSR 133中定义了哪些happen-before规则：

（1）、同一个线程中的，前面的操作 happen-before 后续的操作。（即单线程内按代码顺序执行。但是，在不影响在单线程环境执行结果的前提下，编译器和处理器可以进行重排序，这是合法的。换句话说，这一是规则无法保证编译重排和指令重排）。

（2）、监视器上的解锁操作 happen-before 其后续的加锁操作。（Synchronized 规则）

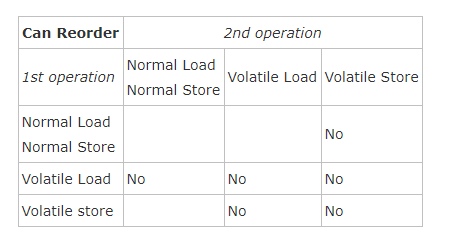
（3）、对volatile变量的写操作 happen-before 后续的读操作。（volatile 规则）

（4）、线程的start() 方法 happen-before 该线程所有的后续操作。（线程启动规则）

（5）线程所有的操作 happen-before 其他线程在该线程上调用 join 返回成功后的操作。

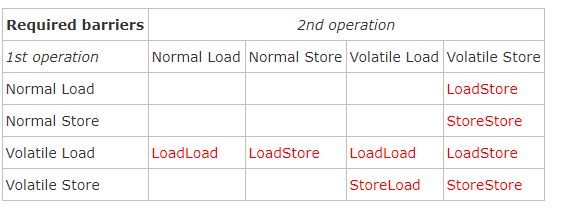
（6）如果 a happen-before b，b happen-before c，则a happen-before c（传递性）。

这里我们主要看下第三条：volatile变量的保证有序性的规则。[《Java并发编程：核心理论》](http://www.cnblogs.com/paddix/p/5374810.html)一文中提到过重排序分为编译器重排序和处理器重排序。为了实现volatile内存语义，JMM会对volatile变量限制这两种类型的重排序。下面是JMM针对volatile变量所规定的重排序规则表：



**3、内存屏障**

　　为了实现volatile可见性和happen-befor的语义。JVM底层是通过一个叫做“内存屏障”的东西来完成。内存屏障，也叫做内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。下面是完成上述规则所要求的内存屏障：



（1）LoadLoad 屏障  
执行顺序：Load1—>Loadload—>Load2  
确保Load2及后续Load指令加载数据之前能访问到Load1加载的数据。

（2）StoreStore 屏障  
执行顺序：Store1—>StoreStore—>Store2  
确保Store2以及后续Store指令执行前，Store1操作的数据对其它处理器可见。

（3）LoadStore 屏障  
执行顺序： Load1—>LoadStore—>Store2  
确保Store2和后续Store指令执行前，可以访问到Load1加载的数据。

（4）StoreLoad 屏障  
执行顺序: Store1—> StoreLoad—>Load2  
确保Load2和后续的Load指令读取之前，Store1的数据对其他处理器是可见的。

### 总结

总体来说，volatile是并发编程中的一种优化，在某些场景下可以代替Synchronized。但是，volatile的不能完全取代Synchronized的位置，只有在一些特殊的场景下，才能适用volatile。总的来说，必须同时满足下面两个条件才能保证在并发环境的线程安全：

（1）对变量的写操作不依赖于当前值。

（2）该变量没有包含在具有其他变量的不变式中。

## transient

我们都知道一个对象只要实现了Serilizable接口，这个对象就可以被序列化，java的这种序列化模式为开发者提供了很多便利，我们可以不必关系具体序列化的过程，只要这个类实现了Serilizable接口，这个类的所有属性和方法都会自动序列化。

 然而在实际开发过程中，我们常常会遇到这样的问题，这个类的有些属性需要序列化，而其他属性不需要被序列化，打个比方，如果一个用户有一些敏感信息（如密码，银行卡号等），为了安全起见，不希望在网络操作（主要涉及到序列化操作，本地序列化缓存也适用）中被传输，这些信息对应的变量就可以加上transient关键字。换句话说，这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化。

总之，java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中

# 操作符

## a = a + b 与 a += b

当a和b的类型不一样时且a精度小于b时，a=a+b编译报错而a+=b自动进行类型转换，将高精度类型强转为定精度类型再进行运算

# HashMap

## 特性

HashMap是非synchronized，所以性能比较高； HashMap储存的是键值对，可以接受null的键和值。

## 原理

HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。

由于hashmap内部的bucket是通过链表形式存放键值对entry，当有两个不同键的hashcode值一样，这两个键对应的键值对entry可以依次添加到该链表中。我们在查找指定键对应的值首先根据键计算出哈希值，找出对应的bucket位置，然后调用keys.equals()方法去找到链表中正确的节点。

如果HashMap的大小超过了负载因子(load factor)定义的容量将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中；重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。

## HashMap 和 HashTable、ConcurrentHashMap 的区别

### ****HashTable****

底层数组+链表实现，无论key还是value都**不能为null**，线程**安全**，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化

初始size为**11**，扩容：newsize = olesize\*2+1

计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

### HashMap

底层数组+链表实现，可**以存储null键和null值**，线程**不安全**

初始size为**16**，扩容：newsize = oldsize\*2，size一定为2的n次幂

扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入

插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）

当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀

计算index方法：index = hash & (tab.length – 1)

### ****ConcurrentHashMap****

底层采用分段的数组+链表实现，线程**安全**

通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)

Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

### 总结

Hashtable和HashMap都实现了Map接口，但是Hashtable的实现是基于Dictionary抽象类的

Hashtable与HashMap另一个区别是HashMap的迭代器（Iterator）是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。

## HashMap、LinkedMap、TreeMap的区别

### 区别

HashMap 是一个最常用的Map，它根据键的HashCode 值存储数据，根据键可以直接获取它的值，具有很快的访问速度。遍历时，取得数据的顺序是完全随机的。  
　　HashMap最多只允许一条记录的键为Null；允许多条记录的值为 Null。  
　　HashMap不支持线程的同步（即任一时刻可以有多个线程同时写HashMap），可能会导致数据的不一致。如果需要同步，可以用 Collections的synchronizedMap方法使HashMap具有同步的能力，或者使用ConcurrentHashMap。  
　　Hashtable与 HashMap类似，它继承自Dictionary类。不同的是：它不允许记录的键或者值为空；它支持线程的同步（即任一时刻只有一个线程能写Hashtable），因此也导致了 Hashtable在写入时会比较慢。

LinkedHashMap保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的。也可以在构造时带参数，按照应用次数排序。  
在遍历的时候会比HashMap慢，不过有种情况例外：当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢。因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。

TreeMap实现SortMap接口，能够把它保存的记录根据键排序。  
　　默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

### 使用场景

1、一般情况下，我们用的最多的是HashMap。HashMap里面存入的键值对在取出的时候是随机的，它根据键的HashCode值存储数据，根据键可以直接获取它的值，具有很快的访问速度。在Map 中**插入、删除和定位元素**，HashMap 是最好的选择。  
　　2、TreeMap取出来的是排序后的键值对。但如果您要**按自然顺序或自定义顺序遍历键**，那么TreeMap会更好。  
　　3、LinkedHashMap 是HashMap的一个子类，如果需要**输出的顺序和输入的相同**,那么用LinkedHashMap可以实现,它还可以按读取顺序来排列，像连接池中可以应用。

### 其他

1. HashSet是通过HashMap实现的,TreeSet是通过TreeMap实现的,只不过Set用的只是Map的key  
2. Map的key和Set都有一个共同的特性就是集合的唯一性.TreeMap更是多了一个排序的功能.  
3. hashCode和equal()是HashMap用的, 因为无需排序所以只需要关注定位和唯一性即可.  
　　a. hashCode是用来计算hash值的,hash值是用来确定hash表索引的.  
　　b. hash表中的一个索引处存放的是一张链表, 所以还要通过equal方法循环比较链上的每一个对象才可以真正定位到键值对应的Entry.  
　　c. put时,如果hash表中没定位到,就在链表前加一个Entry,如果定位到了,则更换Entry中的value,并返回旧value  
4. 由于TreeMap需要排序,所以需要一个Comparator为键值进行大小比较.当然也是用Comparator定位的.  
　　a. Comparator可以在创建TreeMap时指定  
　　b. 如果创建时没有确定,那么就会使用key.compareTo()方法,这就要求key必须实现Comparable接口.  
　　c. TreeMap是使用Tree数据结构实现的,所以使用compare接口就可以完成定位了.

# WeakHashMap

## 简介

WeakHashMap 继承于AbstractMap，实现了Map接口。  
    和[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)一样，WeakHashMap 也是一个**散列表**，它存储的内容也是**键值对(key-value)映射**，而且**键和值都可以是null**。  
   不过WeakHashMap的**键是“弱键”**。在 WeakHashMap 中，当除了自身有对key的引用外，此key没有其他引用那么此map会自动丢弃此值。注意，如果你没有访问过这个对象，即使没有其他引用这个对象，也不会被gc释放。  
    这个“弱键”的原理呢？大致上就是，**通过WeakReference和ReferenceQueue实现的**。 WeakHashMap的key是“弱键”，即是WeakReference类型的；ReferenceQueue是一个队列，它会保存被GC回收的“弱键”。实现步骤是：  
    (01) 新建WeakHashMap，将“**键值对**”添加到WeakHashMap中。  
           实际上，WeakHashMap是通过数组table保存Entry(键值对)；每一个Entry实际上是一个单向链表，即Entry是键值对链表。  
   (02) 当**某“弱键”不再被其它对象引用**，并**被GC回收**时。在GC回收该“弱键”时，**这个“弱键”也同时会被添加到ReferenceQueue(queue)队列**中。  
   (03) 当下一次我们需要操作WeakHashMap时，会先同步table和queue。table中保存了全部的键值对，而queue中保存被GC回收的键值对；同步它们，就是**删除table中被GC回收的键值对**。  
   这就是“弱键”如何被自动从WeakHashMap中删除的步骤了。

和HashMap一样，WeakHashMap是不同步的。可以使用 Collections.synchronizedMap 方法来构造同步的 WeakHashMap

# Hash

## Hashcode作用

HashCode的存在主要是为了查找的快捷性，HashCode是用来在散列存储结构中确定对象的存储地址的

## 一致性哈希算法

一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP使用的简 单哈希算法带来的问题，使得分布式哈希（DHT）可以在P2P环境中真正得到应用。

 一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：

1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

在分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的hash(object)%N算法，那么在有机器添加或者删除后，很多原有的数据就无法找到了，这样严重的违反了单调性原则。接下来主要讲解一下一致性哈希算法是如何设计的：

# List, Set, Map三个接口，存取元素时各有什么特点

List与Set都是**单列元素的集合**，它们有一个功共同的父接口Collection。

**Set**里面不允许有重复的元素，

**存元素：**add方法有一个boolean的返回值，当集合中没有某个元素，此时add方法可成功加入该元素时，则返回true；当集合含有与某个元素equals相等的元素时，此时add方法无法加入该元素，返回结果为false。

**取元素**：没法说取第几个，只能以Iterator接口取得所有的元素，再逐一遍历各个元素。

**List**表示有先后顺序的集合，

**存元素：**多次调用add(Object)方法时，每次加入的对象按先来后到的顺序排序，也可以插队，即调用add(int index,Object)方法，就可以指定当前对象在集合中的存放位置。

**取元素：**方法1：Iterator接口取得所有，逐一遍历各个元素

        方法2：调用**get(index i)**来明确说明取第几个。

**Map**是**双列的集合**，存放用put方法:put(obj key,obj value)，每次存储时，要存储一对key/value，不能存储重复的key，这个重复的规则也是按equals比较相等。

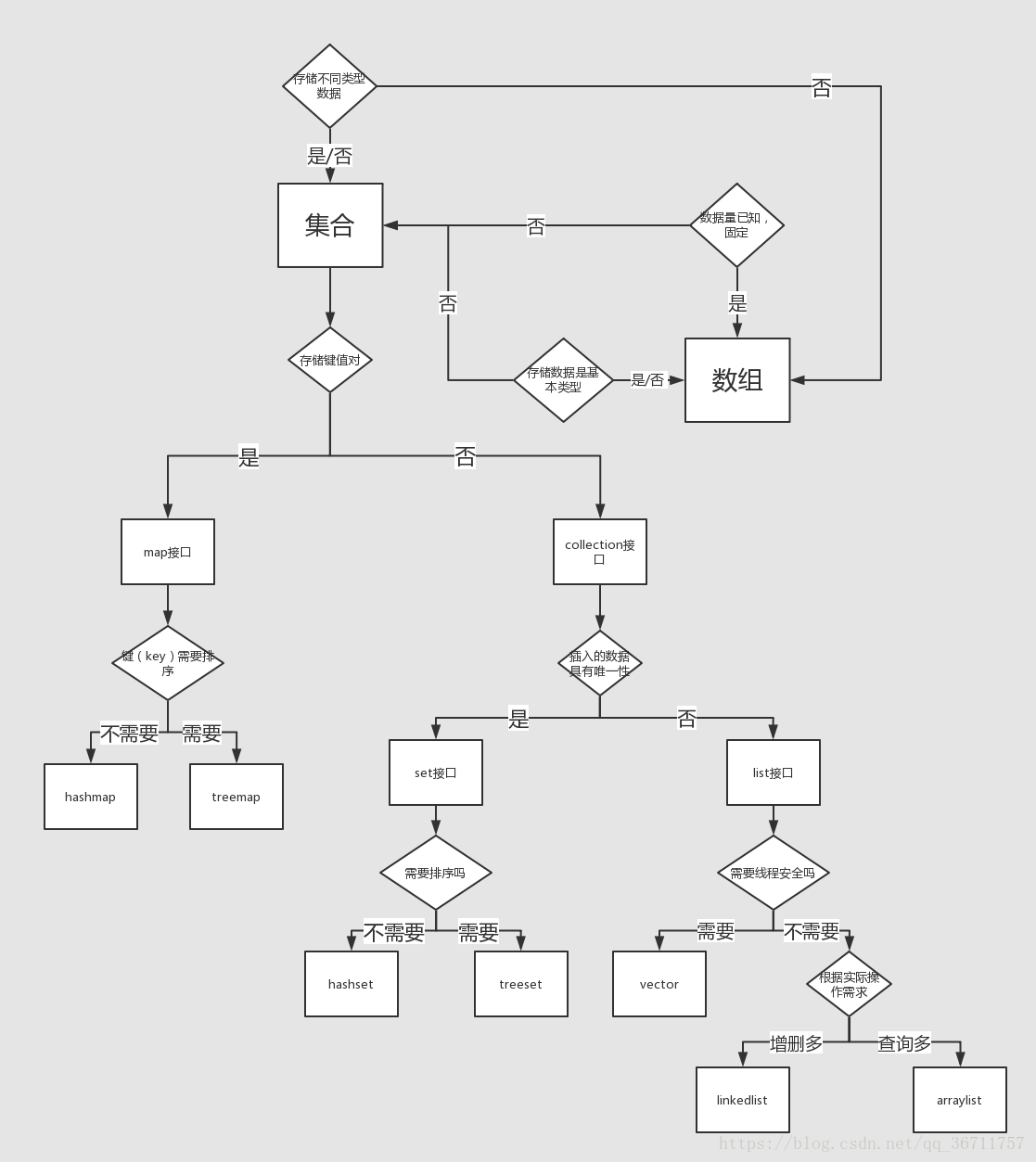
**取元素：**用get(Object key)方法根据key获得相应的value。

        也可以获得所有的key的集合，还可以获得所有的value的集合，

        还可以获得key和value组合成的Map.Entry对象的集合。

List以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 保存key-value值，value可多值。

# 集合框架



# 拷贝Clone

## 如何实现拷贝

1、实现Cloneable接口并重写Object类中的clone()方法；

（1）实现Cloneable接口   
（2）重载Object类中的clone()方法，重载时需定义为public   
（3）在重载方法中，调用super.clone()

2、实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆；

# GC

## 内存分代

### 内存划分

**新生代**

新生代也称Young区，eden区也称伊甸园（《圣经》中亚当和夏娃出生的地方），很形象的比喻对象的出生地点，全部的新生对象都会出现在eden区。s0和s1空间合称幸存者空间（Survivor Spaces），至于什么作用，后续介绍。

新生代的绝大部分对象有朝生熄灭的特点，存活率很低，回收效率很高。

**老年代**

老年代也称Old区。老年代的对象是由新生代中存活多次，或者是特殊原因（原因有多种，后续介绍）从新生代直接转移来的。老年代中对象的生命周期较长，垃圾回收后对象存活率较高，但是回收的效率较慢。

**永久代**

永久代也称Permanent区或方法区。存储类信息、常量、静态变量以及即时编译器编译后的代码等等数据。相对而言，垃圾回收行为在这个区域是比较少出现的，但并非数据进入了永久代就如它的名字一样“永久”存在了，这片区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说，这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是必要的。

### 为什么要分代

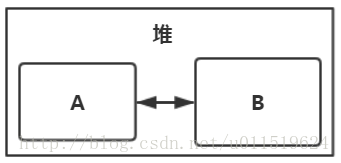
堆内存是虚拟机管理的内存中最大的一块，也是垃圾回收最频繁的一块区域。当然了，我们写的代码中new的对象等都存在堆内存中。如果说堆内存没有区域划分，所有新创建的对象和生命周期很长的对象放在一个区域，随着对象越来越多触发了JVM的垃圾回收机制，而每次回收都要遍历所有的对象，这个时间成本是难以想象的，严重影响GC效率。

而有了内存分代，新创建的对象会在新生代中分配内存，而经过多次回收仍然存活下的对象或者有特殊情况的对象会移至老年代，类信息、静态变量等信息存放在永久代里。新生代中的对象存活时间短，只需要在新生代区域中频繁进行GC，而老年代中对象存活时间长，回收的频率相对就低很多了，永久代则很少进行垃圾回收。所以呢，给堆内存分代是为了提高对象内存分配和垃圾回收的效率。

## 对象存活判断算法

### 引用计数法

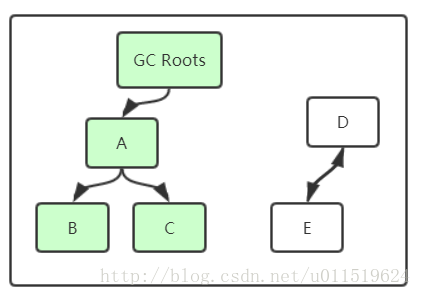
引用计数法就是给对象加个引用计数器，每有一个地方引用到它时，这个引用计数器就加1。当引用失效的时候，计数器的值就减1，也就是说根据引用计数器的值来判断对象是否存活，若值是0，那么该对象就不再被使用了。   
真是如此吗？答案是否定的。试想一下如下图的场景：



AB对象相互引用，那么AB对象的引用计数器的值永远都不会为0，AB就永远都不会被回收，直接造成内存泄漏问题。所以该算法最大的缺点是很难解决对象之间相互引用的问题。但事实上，如上图情况的相互引用对象会被回收，说明实际不是用引用计数法判定对象存活与否。

### 可达性分析法

可达性分析法的基本思路是通过一系列的GC Roots对象作为起始点，从这些点向下搜索它们引用的对象，这样可以生成一颗引用树，树的节点就是可达的对象。反之，不在树上的对象即可判定对象已死。来看如下图：



D和E对象相互引用，但是它们没有RC Roots根节点的引用指向，所以D和E在下次垃圾回收时会被处死，而ABC对象在GC Roots的引用树上，会视为存活对象。

那么哪些对象可以作为引用树的根节点呢？

**可作为GC Roots的对象：**

不难想象，作为GC Roots的对象必须是极难被回收的对象，包括了如下几种对象：

虚拟机栈（栈帧中的本地变量表）中引用的对象，如在方法中定义和使用的变量

方法区中的类静态属性引用的对象，如static修饰的成员变量

方法区中常量引用的对象，如static和final共同修饰的常量

本地方法栈中JNI引用的对象，JNI也就是调的native方法

### 引用类型与垃圾回收机制

不论是通过以上哪种方式判断对象是否存活，都与引用相关。java自1.2之后，对引用划分了4种，如下：

强引用：只要某个对象有强引用与之关联，JVM必定不会回收这个对象，即使在内存不足的情况下，JVM宁愿抛出OutOfMemory错误也不会回收这种对象。

软引用：软引用是用来描述一些有用但并不是必需的对象，比如引用图片地址等，在Java中用java.lang.ref.SoftReference类来表示。对于软引用关联着的对象，只有在内存不足的时候JVM才会回收该对象。

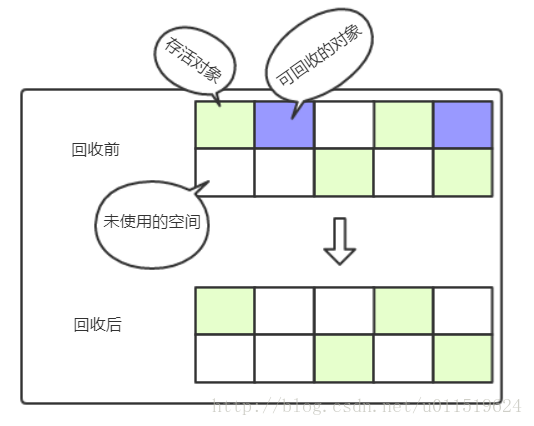
弱引用：弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。也就是说被弱引用关联的对象，只能生存到下一次垃圾收集发生之前。

虚引用（幽灵引用或幻影引用）：虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。与弱引用区别：在GC时会被通知。

## 垃圾回收算法

### 标记-清除算法

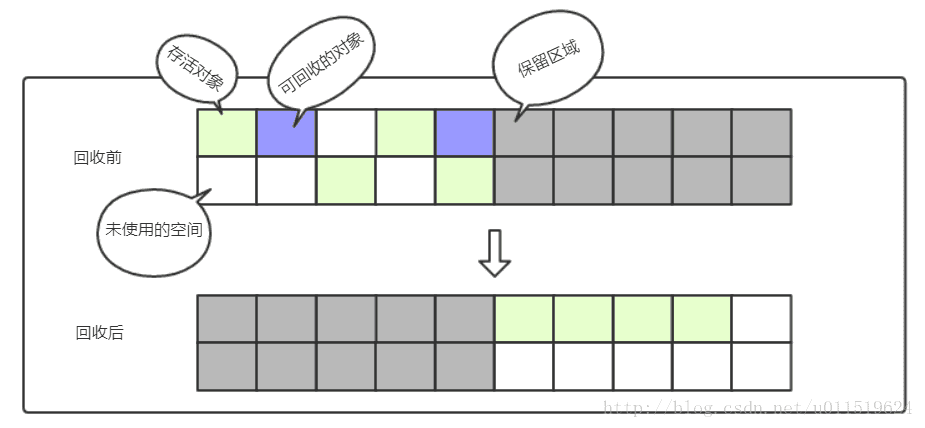
算法的过程就如同算法的名字，分为标记和清除两个阶段：首先标记出所有要回收的对象，然后对标记的对象统一清除。算法很简单，再看下逻辑图：



内存空间经过标记清除算法回收的过程一目了然，从图中也能看出这种算法的不足，即标记清除后会产生大量的不连续的内存碎片，如果碎片过多会导致大对象无法分配足够的空间而不得不触发垃圾回收。   
另外，标记清除算法的标记和清除阶段效率都不高，所以效率也是个问题。

### 复制算法

既然标记清除效率达不到预期，那么新的回收算法就被创造出来了–复制算法。算法将可用内存分成两块大小一样的区域，每次只使用其中的一块区域，当使用的区域空间不足时就得进行垃圾回收了，而存活的对象就直接转移至另一块区域，如此往复。看如下复制算法逻辑图：

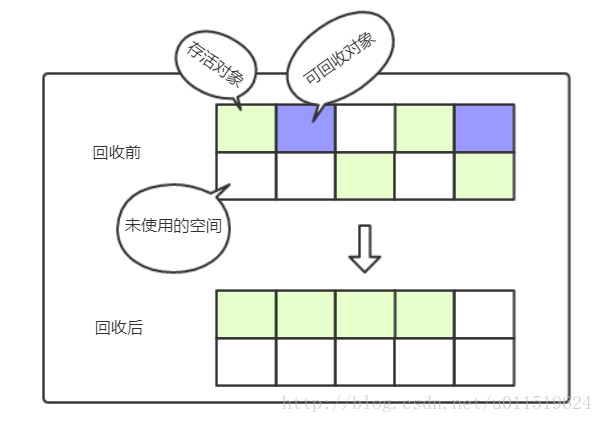


每次垃圾回收都是在一块区域进行，即一半的空间进行内存回收。好处是内存分配后不会产生内存碎片问题，当然缺点也显而易见，内存空间利用率只有一半。

由于每次垃圾回收需要对存活对象再一次迁移，这种算法适用于存活对象较少的场景。根据大多数经验表明，新生代的对象98%都是朝生熄灭的，所以复制算法适应于新生代的垃圾回收。但是在新生代中，一半的空间利用率实在太低，而且绝大部分对象都会被回收的特性，所以将内存分为了较大的eden区和两个较小的survivor区（s0和s1区），每次使用eden区和其中的一个survivor区，另一个survivor区则用于保存存活的对象。HotSpot虚拟机默认eden和survivor的比例是8:1，也就是说空间利用率达到了90%（80%+10%）。

### 标记-整理算法

顾名思义，看名字大致与标记-清除算法类似，其算法的标记过程还真是一样，但后续不是直接清除垃圾对象，而是让所有的存活的对象向一个方向移动，然后再清理掉存活对象边界之外的内存空间。来看图更清晰：



上图与标记清除算法的图一比较就很容易发现区别了，标记整理算法重在整理。

那么标记整理算法相对于复制算法有什么优势呢？复制算法，在回收对象存活率低的情况下比较合适，如果在对象存活率较高时（比如老年代的垃圾回收），就要进行较多的复制操作，效率明显变低，所以不会采用复制算法。而在对象存活较高的区域中，如老年代，采用标记整理算法是比较合适的。

### 分代收集算法

在前面内存分代的文章中说过，堆中根据对象的存活时间划分为新生代、老年代和永久代几个区域，这里又可以根据各个年代的特点采用最合适的垃圾回收算法。新生代绝大部分对象朝生熄灭，只有少部分存活，采用复制算法最合适不过。老年代对象即使进行了垃圾回收，对象的存活率也高，所以采用标记清除或标记整理算法都是不错的选择。

### GC过程

一共有三个空间，其中包含两个幸存者空间。每个空间的执行顺序如下：

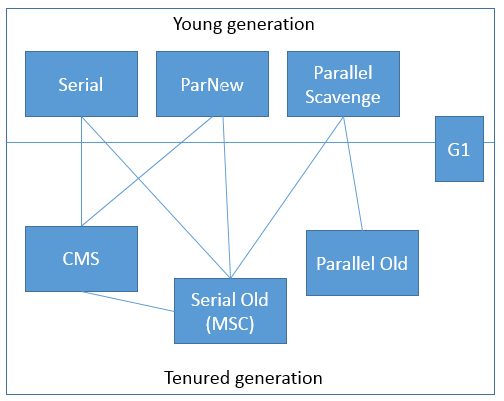
绝大多数刚刚被创建的对象会存放在伊甸园空间。

在伊甸园空间执行了第一次GC之后，存活的对象被移动到其中一个幸存者空间。

  此后，在伊甸园空间执行GC之后，存活的对象会被堆积在同一个幸存者空间。

 当一个幸存者空间饱和，还在存活的对象会被移动到另一个幸存者空间。之后会清空已经饱和的

## 垃圾收集器

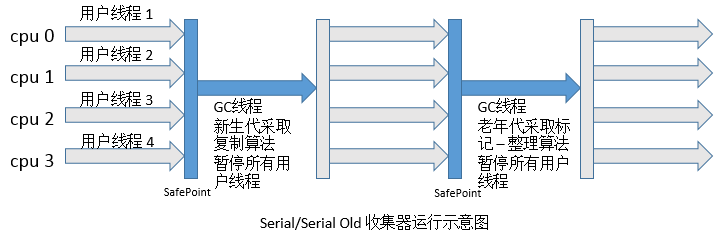


上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，那说明它们可以搭配使用。虚拟机所处的区域说明它是属于新生代收集器还是老年代收集器。多说一句，我们必须明确一个观点：没有最好的垃圾收集器，更加没有万能的收集器，只能选择对具体应用最合适的收集器。这也是HotSpot为什么要实现这么多收集器的原因。OK，下面一个一个看一下收集器

**1、Serial收集器**

单线程收集器，收集时会暂停所有**工作**线程（我们将这件事情称之为Stop The World，下称STW），使用复制收集算法，虚拟机运行在Client模式时的默认新生代收集器。

最基本、发展历史最久的收集器，这个收集器是一个**采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止**。后者意味着，在用户不可见的情况下要把用户正常工作的线程全部停掉，这对很多应用是难以接受的。不过实际上到目前为止，**Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器**，因为它简单而高效。用户桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代停顿时间在几十毫秒最多一百毫秒，只要不是频繁发生，这点停顿是完全可以接受的。Serial收集器运行过程如下图所示：

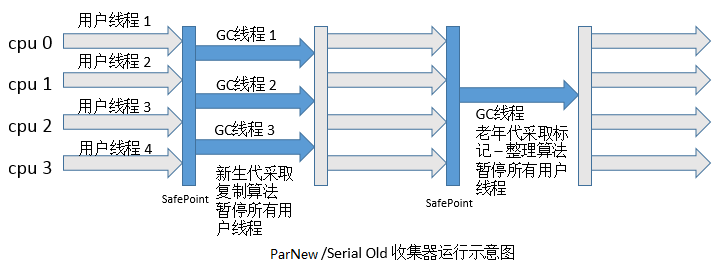


说明：1. 需要STW（Stop The World），停顿时间长。2. 简单高效，对于单个CPU环境而言，Serial收集器由于没有线程交互开销，可以获取最高的单线程收集效率

**2、ParNew收集器**

ParNew收集器就是Serial的多线程版本，除了使用多条收集线程外，其余行为包括算法、STW、对象分配规则、回收策略等都与Serial收集器一摸一样。对应的这种收集器是虚拟机运行在**Server**模式的默认新生代收集器，在单CPU的环境中，ParNew收集器并不会比Serial收集器有更好的效果。

**ParNew收集器其实就是Serial收集器的多线程版本**，除了使用多条线程进行垃圾收集外，其余行为和Serial收集器完全一样，包括使用的也是复制算法。ParNew收集器除了多线程以外和Serial收集器并没有太多创新的地方，**但是它却是Server模式下的虚拟机首选的新生代收集器**，其中有一个很重要的和性能无关的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作（看图）。CMS收集器是一款几乎可以认为有划时代意义的垃圾收集器，因为它第一次实现了让垃圾收集线程与用户线程基本上同时工作。ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于线程交互的开销，该收集器在两个CPU的环境中都不能百分之百保证可以超越Serial收集器。当然，随着可用CPU数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU数量相同，在CPU数量非常多的情况下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。ParNew收集器运行过程如下图所示：



**3、Parallel Scavenge收集器**

     Parallel Scavenge收集器也是一个新生代收集器，也是用复制算法的收集器，也是并行的多线程收集器，但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍**吞吐量**的概念。**CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是打到一个可控制的吞吐量**。所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即**吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）**，虚拟机总运行100分钟，垃圾收集1分钟，那吞吐量就是99%。另外，**Parallel Scavenge收集器是虚拟机运行在Server模式下的默认垃圾收集器**。

     停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；高吞吐量则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。

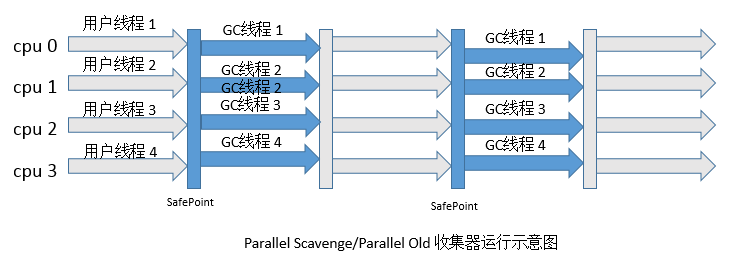
虚拟机提供了-XX:MaxGCPauseMillis和-XX:GCTimeRatio两个参数来精确控制最大垃圾收集停顿时间和吞吐量大小。不过不要以为前者越小越好，GC停顿时间的缩短是以牺牲吞吐量和新生代空间换取的。由于与吞吐量关系密切，**Parallel Scavenge收集器也被称为“吞吐量优先收集器”**。Parallel Scavenge收集器有一个-XX:+UseAdaptiveSizePolicy参数，这是一个开关参数，这个参数打开之后，就不需要手动指定新生代大小、Eden区和Survivor参数等细节参数了，虚拟机会根据当前系统的运行情况手机性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。**如果对于垃圾收集器运作原理不太了解，以至于在优化比较困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择**。

**4、Serial Old收集器**

**Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理算法”**，这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

**5、Parallel Old收集器**

**Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法**。这个收集器在JDK 1.6之后的出现，“吞吐量优先收集器”终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge收集器+Parallel Old收集器的组合。运行过程如下图所示：



**6、CMS收集器**

**CMS（Conrrurent Mark Sweep）收集器是以获取最短回收停顿时间为目标的收集器。使用标记 - 清除算法**，收集过程分为如下四步：

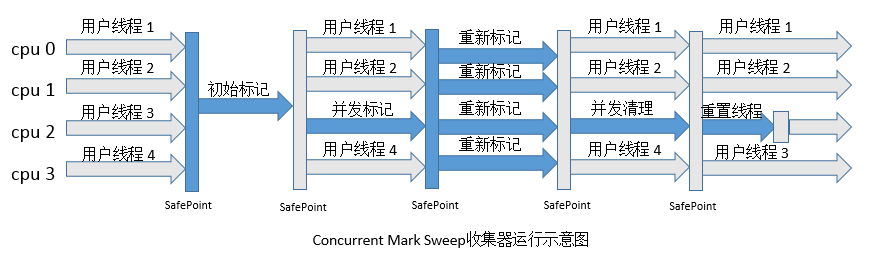
(1). 初始标记，标记GCRoots能直接关联到的对象，时间很短。

(2). 并发标记，进行GCRoots Tracing（可达性分析）过程，时间很长。

(3). 重新标记，修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，时间较长。

(4). 并发清除，回收内存空间，时间很长。

其中，并发标记与并发清除两个阶段耗时最长，但是可以与用户线程并发执行。运行过程如下图所示:



说明：1. 对CPU资源非常敏感，可能会导致应用程序变慢，吞吐率下降。2. 无法处理浮动垃圾，因为在并发清理阶段用户线程还在运行，自然就会产生新的垃圾，而在此次收集中无法收集他们，只能留到下次收集，这部分垃圾为浮动垃圾，同时，由于用户线程并发执行，所以需要预留一部分老年代空间提供并发收集时程序运行使用。3. 由于采用的标记 - 清除算法，会产生大量的内存碎片，不利于大对象的分配，可能会提前触发一次Full GC。虚拟机提供了-XX:+UseCMSCompactAtFullCollection参数来进行碎片的合并整理过程，这样会使得停顿时间变长，虚拟机还提供了一个参数配置，-XX:+CMSFullGCsBeforeCompaction，用于设置执行多少次不压缩的Full GC后，接着来一次带压缩的GC。

**7、G1收集器**

G1是目前技术发展的最前沿成果之一，HotSpot开发团队赋予它的使命是未来可以替换掉JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1收集器有以下特点：

(1). 并行和并发。使用多个CPU来缩短Stop The World停顿时间，与用户线程并发执行。

(2). 分代收集。独立管理整个堆，但是能够采用不同的方式去处理新创建对象和已经存活了一段时间、熬过多次GC的旧对象，以获取更好的收集效果。

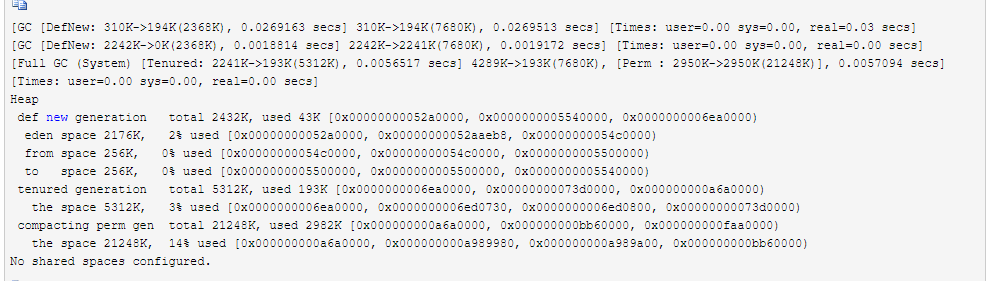
(3). 空间整合。基于标记 - 整理算法，无内存碎片产生。

(4). 可预测的停顿。能简历可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

     在G1之前的垃圾收集器，收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分（可以不连续）Region的集合。

## GC日志

每种收集器的日志形式都是由它们自身的实现所决定的，换言之，每种收集器的日志格式都可以不一样。不过虚拟机为了方便用户阅读，将各个收集器的日志都维持了一定的共性，来看下面的一段GC日志：



1、日志的开头“GC”、“Full GC”表示这次垃圾收集的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有Full，则说明本次GC停止了其他所有工作线程(Stop-The-World)。看到Full GC的写法是“Full GC(System)”，这说明是调用System.gc()方法所触发的GC。

2、“GC”中接下来的“[DefNew”表示GC发生的区域，这里显示的区域名称与使用的GC收集器是密切相关的，例如上面样例所使用的Serial收集器中的新生代名为“Default New Generation”，所以显示的是“[DefNew”。如果是ParNew收集器，新生代名称就会变为“[ParNew”，意为“Parallel New Generation”。如果采用Parallel Scavenge收集器，那它配套的新生代称为“PSYoungGen”，老年代和永久代同理，名称也是由收集器决定的。

3、后面方括号内部的“310K->194K(2368K)”、“2242K->0K(2368K)”，指的是**该区域已使用的容量->GC后该内存区域已使用的容量(该内存区总容量)**。方括号外面的“310K->194K(7680K)”、“2242K->2241K(7680K)”则指的是**GC前Java堆已使用的容量->GC后Java堆已使用的容量(Java堆总容量)**。

4、再往后“0.0269163 secs”表示该内存区域GC所占用的时间，单位是秒。最后的“[Times: user=0.00 sys=0.00 real=0.03 secs]”则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的墙钟时间。后面两个的区别是，墙钟时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间，所以如果看到user或sys时间超过real时间是完全正常的。

5、“Heap”后面就列举出堆内存目前各个年代的区域的内存情况。

# JVM

## JVM参数设置

不管是YGC还是Full GC,GC过程中都会对导致程序运行中中断,正确的选择[不同的GC策略](http://www.cnblogs.com/redcreen/archive/2011/05/04/2037029.html),调整JVM、GC的参数，可以极大的减少由于GC工作，而导致的程序运行中断方面的问题，进而适当的提高Java程序的工作效率。但是调整GC是以个极为复杂的过程，由于各个程序具备不同的特点，如：web和GUI程序就有很大区别（Web可以适当的停顿，但GUI停顿是客户无法接受的），而且由于跑在各个机器上的配置不同（主要cup个数，内存不同），所以使用的GC种类也会不同(如何选择见[GC种类及如何选择](http://www.cnblogs.com/redcreen/archive/2011/05/04/2037029.html))。本文将注重介绍JVM、GC的一些重要参数的设置来提高系统的性能。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数名称** | **含义** | **默认值** |  |
| -Xms | 初始堆大小 | 物理内存的1/64(<1GB) | 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制. |
| -Xmx | 最大堆大小 | 物理内存的1/4(<1GB) | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制 |
| Xmn | 年轻代大小(1.4or lator) |  | **注意**：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小. 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官 |
| -XX:NewSize | 设置年轻代大小(for 1.3/1.4) |  |  |
| -XX:MaxNewSize | 年轻代最大值(for 1.3/1.4) |  |  |
| -XX:PermSize | 设置持久代(perm gen)初始值 | 物理内存的1/64 |  |
| -XX:MaxPermSize | 设置持久代最大值 | 物理内存的1/4 |  |
| -Xss | 每个线程的堆栈大小 |  | JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。（校长） 和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"” -Xss is translated in a VM flag named ThreadStackSize” 一般设置这个值就 |
| -XX:ThreadStackSize | Thread Stack Size |  |  |
| -XX:NewRatio | 年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代) |  | -XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5 Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。 |
| -XX:SurvivorRatio | Eden区与Survivor区的大小比值 |  | 设置为8,则两个Survivor区与一个Eden区的比值为2:8 |
| -XX:LargePageSizeInBytes | 内存页的大小不可设置过大， 会影响Perm的大小 |  |  |
| -XX:+UseFastAccessorMethods | 原始类型的快速优化 |  |  |
| -XX:+DisableExplicitGC | 关闭System.gc() |  | 这个参数需要严格 |

**并行收集器相关参数**

|  |  |  |  |
| --- | --- | --- | --- |
| -XX:+UseParallelGC | Full GC采用parallel MSC (此项待验证) |  | 选择垃圾收集器为并行收集器.此配置仅对年轻代有效.即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收 |
| -XX:+UseParNewGC | 设置年轻代为并行收集 |  | 可与CMS收集同时使用 JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值 |
| -XX:ParallelGCThreads | 并行收集器的线程数 |  | 此值最好配置与处理器数目相等 同样适用于CMS |
| -XX:+UseParallelOldGC | 年老代垃圾收集方式为并行收集(Parallel Compacting) |  | 这个是JAVA 6出现 |
| -XX:MaxGCPauseMillis | 每次年轻代垃圾回收的最长时间(最大暂停时间) |  | 如果无法满足此时间,JVM会自动调整年轻代大小,以满足此值. |
| -XX:+UseAdaptiveSizePolicy | 自动选择年轻代区大小和相应的Survivor区比例 |  | 设置此选项后,并行收集器会自动选择年轻代区大小和相应的Survivor区比例,以达到目标系统规定的最低相应时间或者收集频率等,此值建议使用并行收集器时,一直打 |
| -XX:GCTimeRatio | 设置垃圾回收时间占程序运行时间的百分比 |  | 公式为1/(1+n) |
| -XX:+ScavengeBeforeFullGC | -XX:+ScavengeBeforeFullGC | true |  |

# 性能调优