Java基础

# Java数据类型与操作符

## 基本数据类型

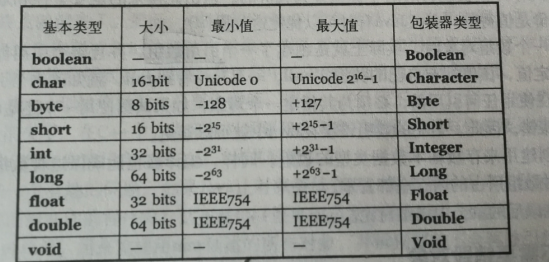


图1

如图1，基本数据类型一种有七种，其中需要注意的是void也是一种，每一个基本类型都有对应的包装器类型。

基本数据类型不是用new创建出来的，而是创建一个非引用的自动变量，这个变量直接存储“值”，并置于堆栈中，效率更加高。包装器类的作用可以在堆中创建一个非基本对象，用来表示对应的基本类型。

## 数组

在java中，数组是会被强制初始化，而且不能在它的范围之外进行访问，当创建了一个数组的时候，实际上是创建了一个引用数组，该引用被初始化为Null。

## static关键字

当声明一个事物是static时，就意味着这个域或方法不会与包含它的那个类的任何对象实例关联在一起，也就是说，不需要对象来对其进行访问，类就可以进行访问。被static修饰的域或方法只会创建一次。

对于static方法，其内部不可以调用非静态的方法，反过来非静态的方法可以调用静态方法。

同时，static里面不可以调用this，并且可以没有创建对象的情况下，仅仅通过类本身就可以对其进行访问，

## 移位操作符

移位操作符操作的预算对象是二进制的‘位’，移位操作符只可用来处理整数类型的数据。左移位操作符（<<）能按照操作符右侧指定的数位将整数操作符左边的操作数向左移动（在低位补0），“有符号”的右移操作符（>>）则按照操作符右边指定的数位将整数操作符左边的操作数向右移动。“有符号”右移位操作符使用“符号扩展”：若符号为正，则在高位插入0，否则在高位插入1。Java中增加了一种无符号右移操作符（>>>），它使用“**零拓**展”：无论正负，都在高位插0。

## 截尾与舍入

float f = 0.7

int a = (int)f

其最终的结果是多少？其实最终的结果是0，因为在向下类型转换的时候，Java中的处理总是对该数字进行截取操作。如果想要四舍五入，那么就需要使用Math中的round()方法。

# Java初始化与清理

## 构造器初始化

构造器在为对象分配存储空间之后，会调用相应的构造器，确保了在使用对象的时候能够恰当的初始化。

默认构造器不接受任何参数。如果创建了一个类，没有指定构造器，那么，编译器会帮你创建一个默认的构造器。如果说，类里面定义了其他带参数的构造器，编译器就不会帮你创建一个默认构造器，那么只可以使用那些带参数的构造器，而不可以使用默认构造器，如果说想要使用默认构造器，那么就需要自己显式的声明。

## 方法的重载

重载就是让方法名相同，而参数列表中的参数不相同，且方法名相同的方法可以同时存在。

Java中区分重载的方法就是看方法的形参列表，每个重载的方法都有一个独一无二的形参列表，以至于形参的顺序不相同，也可以区分两个方法。但是不可以根据返回值的不同来区分重载的不同的方法，因为返回值不一定会用到。

基本数据类型的重载需要特别注意如下，我们有7个重载方法：

1. f(char x)
2. f(byte x)
3. f(short x)
4. f(int x)
5. f(long x)
6. f(float x)
7. f(double x)

* 如果我们传入常量5，默认的是选择f(int x )这个重载方法，如果没有则会优先选择最近的类型进行提升。如果只有1-3，那么就会报错，如图2，图3所示。对于常量，默认的就是int。



图2

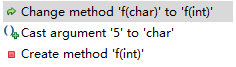


图3

* 如果传入的是char，默认的是选择f(char)，没有就会提升到int及以上的类型，不会提升到short或者byte类型。
* 如果传入的是byte，默认的选择是f(byte)，没有就会选择与之最接近的一个类型进行提升。Short，int，long，float，double，跟byte相似。

如果传入的类型较大，那么就需要进行强制类型转换。

## this关键字

this关键字代指的是当前对象的一个引用，this关键字只能够在方法的内部使用。例如有类Banana，并创建对象banana，其中包含一个方法peel，可以通过创建的对象来调用banana.peel(1)，实际上内部的表现形式是

Banana.peel(banana,1)，编译器默认会把调用该方法的对象作为隐形参数传入到调用的方法中，而我们获取可以通过this关键字来获取。

this关键字还可以再构造器中调用构造器。但是不可以同时调用两次，而且，this关键字调用构造器的时候必须放在最起始处，否则会报错。

## 清理与垃圾回收

### finalize方法

finalize方法主要用于处理那些非创建对象方式为对象分配内存的情况，例如C和C++中的malloc()方法，malloc分配后的内存需要通过free方法来释放，这就需要在finalize方法中进行操作。

finalize方法在垃圾回收器准备释放内存的时候，将首先调用finalzie方法，此时通过new创建的对象并不会被释放，直到垃圾回收器真正回收的时候才会去回收new的对象，也就是说，finalize是在垃圾回收器之前进行操作。

### 垃圾回收器工作

Pass

### 初始化顺序

比方说，我们创建了一个对象：

1. 首先，我们必须要保证成员变量得到恰当的初始化，也就是说，成员变量会自动的进行初始化，并且在构造函数、静态块，构造块之前进行初始化。
2. 在成员变量初始化之后，就会轮到静态代码块初始化，静态代码块只会初始化一次，下次创建对象的时候就不会在调用静态块，静态块的目的主要是用于初始化静态成员变量。静态代码块的初始化顺序按照代码的顺序。
3. 之后就是构造块的初始化了，构造块的目的就是为了初始化非静态成员变量。
4. 之后才轮到构造函数的调用，这样的步骤之后确保了构造函数里面调用的成员变量得到了恰当的初始化。

需要说明的一点就是，静态初始化只有在必要的时候才能够创建，比如创建一个对象，或者直接通过类调用静态成员变量或则方法，这种时候才会执行静态初始化。并且只会执行一次。

### final关键字

final关键字意为最终的意思，通常表示无法更改。

* final数据：

经过final修饰过的成员变量要么在定义的时候就进行初始化，要么空着，让其在构造器当中进行初始化。对于基本数据类型，一旦初始化之后，其值是无法进行更改的。而对于引用之前加了final，则引用无法指向另外一个对象，但是他所指向的对象的内容是可以改变的。

final与static结合就可以表示为常量，一般常量使用大写字母表示，字符与字符之间使用\_进行分割。

* final参数：

final参数是指在参数列表中将参数申明为final类型，为了不让用户修改传入参数的值，具体细节如上述的final数据所示。

* final方法:

方法声明为final的目的第一是为了不让子类对他进行继承，无法修改其含义。第二是为了效率，不过随着Java版本的迭代，后者越发不重要。

# 接口与内部类

## 抽象

创建抽象类的目的是希望通过这个通用的接口来操纵一系列的类。抽象类里面不一定包含抽象方法，但是包含抽象方法的类一定是抽象类。抽象类被子类继承，如果子类未实现抽象方法，那么子类也是抽象类，否则子类就不是抽象类。

## 接口

接口就是一个完全抽象的类，里面是方法都是抽象方法，接口被用来建立类与类之间的协议。接口可以包含域，但是这些域隐式地是static和final。并且接口中的方法都是public。

接口可以用来做多重继承，Java中一个类只可以继承一次，但是却可以实现多个接口。保证了类既像这个，又像那个。可以保证向上转型为每一个接口。

如果父类实现了一个接口，那么子类继承父类，也会继承该接口，并且可以调用该接口的方法。除此之外，接口之间也是可以继承，通过关键字extends即可继承。继承后的接口，如果子类实现它，那么就需要实现接口当中的所有方法，包括继承过来了。

## 内部类

内部类，顾名思义，就是在类、方法、作用域中定义的类，内部类拥有对外围类所有成员的访问权限，内部类中存储着一个外部类引用。外部类也可以访问内部类。如果在内部类中想要使用一个外部类对象的引用，那么就需要使用类名.this来调用内部类的引用。如果想要创建一个内部类，那么就需要使用.new，具体语法如下：Outer.inner oi = outerinstance.new Inner()；内部类中不可以有static数据以及static字段。

匿名内部类是一种常用的形式，具体形式如下：

New Interface {

};

这样的类没有名字，其含义是该对象是继承于Interface的一个对象，并且向上转型成接口类型。

上述的匿名内部类使用的是，默认构造器，如果基类需要一个有参数的构造器，那么可以这么写:

New Interface(parameter param) {

};

上述并非是匿名内部类需要一个参数，而是直接将参数传递给了基类，这样基类就可以得到初始化，就可以使用基类当中的方法。如内部类也像要进行一定初始化，那么可以这么做：

New Interface(parameter param) {

{

Initial；

}

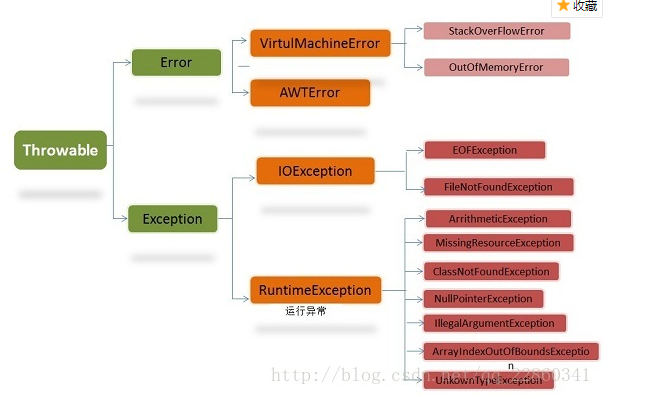
};

这样就可以初始化，但是构造器不能重载，因为就只有一个构造器。

还需要注意的一点就是，如果一个匿名内部类需要使用在其外部定义的一个对象，那么编译器就要求内部类中的参数引用必须是final类型。

内部类当中的是不可以声明静态方法以及静态域的。创建outter的时候，Inner的class是不会被加载到，那么也就无法使用静态方法与变量。如果外部类中有一个静态方法，想要创建内部类对象，那么也是无法进行创建，理由与上面的相同。

# 异常



异常继承体系如上图所示。异常是一种处理错误的方式，Java中所有的异常都是继承自Throwable，意为可抛异常。异常中需要特别注意的一个异常是RuntimeException，运行是异常Java虚拟机会自己进行抛出，可以不需要进行抛出异常的声明。

Java中，异常可能会丢失，丢失的两种情况分别为

1. 在finally中抛出异常。
2. 在finally中调用return。

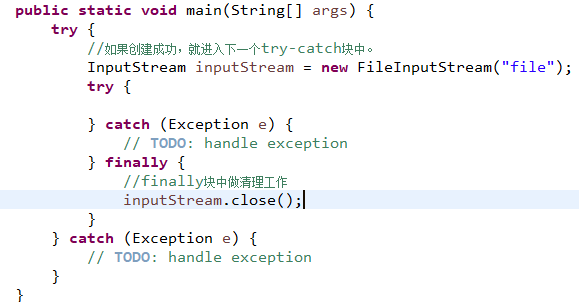
上述的丢失是指没有使用catch的情况下。

除此之外，在继承过程中，异常有很多限制

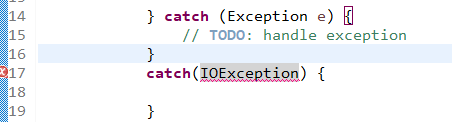
1. 如果基类方法存在一个方法于接口中的方法名称一样，那么子类继承实现他们的时候，该方法要么抛出共有的异常，要么抛异常。
2. 对于构造器，异常可以不跟基类的相同，但是必须包含基类。如果创建的对象没有向上转型，那么就抛子类的异常，如果向上转型了，那么就需要抛出基类的异常。
3. 对于覆写的方法，子类抛出的异常可以跟父类一样，或者是父类异常的子类，或者不抛异常。

总的来说，异常说明接口随着继承越来越小。

一般来说，一个安全的异常写法需要使用嵌套语句，例如，创建对象的时候会抛异常，这个时候就需要一个套上一层try-catch，不需要finally进行清理，如果创建成功，那么就来一个try-catch-finally语句，如果此时抛出异常，那么就不是创建的异常，创建的对象已经成功，不管怎样，需要在finally中对该对象进行清理。具体例子如下：



还有一个需要注意的是异常匹配，异常处理系统会按照代码的书写顺序找出“最近”的来进行处理，当找到一个后，就不需要再往下进行寻找了。派生类的异常可以匹配基类的异常，也就是说，只需要写一个基类，那么派生类就可以被捕捉。但有一种情况需要特别注意，当基类写在派生类之前的时候，编译器就会报错。



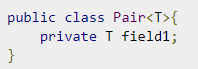
# 泛型

泛型机制将类型转换时的类型检查从运行时提前到了编译时，使用泛型编写的代码比杂乱的使用object并在需要时再强制类型转换的机制具有更好的可读性和安全性。

使用泛型，有这样几个优点：

1. 可读性，从字面上就可以判断集合中内容类型。
2. 类型检查，避免插入非法类型。
3. 获取数据时不再需要强制类型转换。

泛型类的定义方法如下：



类名后面添加<T>，就代表着该类是一个泛型类，field1的类型就跟T的类型是一样的。

泛型方法的定义如下：



其中<T>就是泛型，泛型方法可以放在非泛型类当中，一个严格的泛型方法调用应该是这样的



需要指出泛型的类型。当然，更为简洁的方法是可以省去<String>这样的声明，直接根据参数类型来推断出最终的，例如，根据参数“213”，“result”，“12”可以推断出他们是String类型，这样就可以得出泛型的类型。

需要区分的是泛型方法与泛型类方法，如果一个方法在修饰符跟返回值之间有<T>，那么无论该方法所在的类是不是泛型，该方法的泛型都是独立的，不受影响，否则，就会受到类泛型的影响。所以有<T>的称为泛型方法，没有的称为泛型类中的成员方法。

泛型可以对其进行限定，例如<T extends Comparable>指出该泛型是Comparable的子类。泛型限定可以确保安全，而且可以使用限定类型的方法。

泛型只在编译阶段有效，编译后类型被擦除了，也就是说jvm中没有泛型对象，只有普通对象。所以完全可以把代码编译为jdk1.0运行的字节码。如果说直接定义的<T>，那么就擦除到Object，如果有类型的限定，那么就擦出到限定类型。例如：

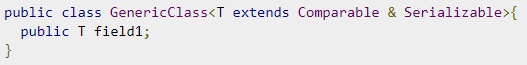


擦除后：



类型T就被Comparable替换。

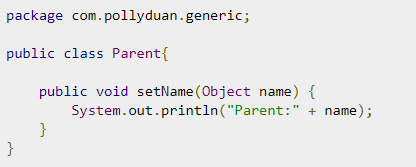
如果有多个限定符，那么就擦除到第一个即可，例如：



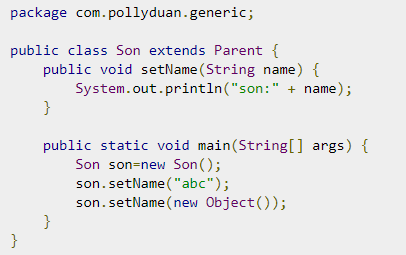
那么擦除后T就使用Comparable代替。如果使用到后面类型的方法时，编译器就会自动插入强制类型转换。

当然，泛型也是有限制的，例如，不能使用8个基本类型实例化类型参数，类型检查不可使用泛型，不能创建泛型对象、数组。也不能在泛型类的静态域中使用泛型类型，但是静态的泛型方法是可以使用泛型类型的。

接下来会有泛型的一些小问题：例如有这么一个父类：

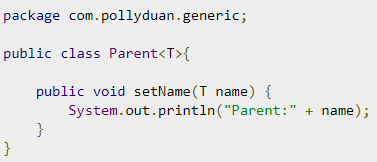


其子类是这样子：

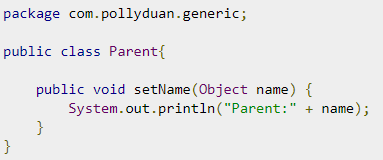


子类中就重载了父类中的setName方法，main方法中的调用也是没有问题的。

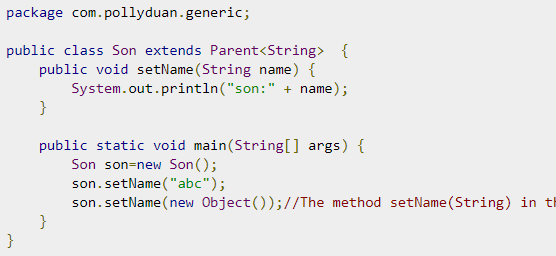
如果改成泛型，其结果如下：



其擦除的结果为：

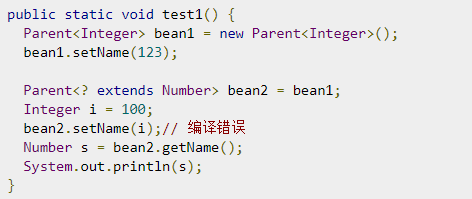


子类是这样：

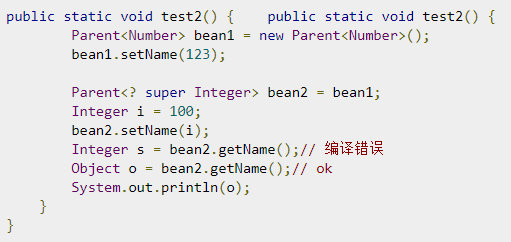


可以发现，之前可以重载的方法，在泛型类中就不可以了，因为在继承的时候，父类的类型已经被确定为String类型，也就不存在那个Object类型的方法了。

通配符类型有两点需要注意：



很显然bean2 可以指向bean1，但是不可以调用setName方法，因为bean2可以指向任何继承自Number的对象类型变量，但是不可以使用setName方法是因为bean2可以指向Integer，Double等方法，那么传入Integer、Double也都是合理的，就如上面所示，bean2指向的是Integer，如果传入一个Double，看上去合理但是却不可行。所以Java干脆不允许泛型的子类型的通配类型作为入参。



而在这个方法中，setName可行，在于Integer的父类可以接收一个Integer参数。而getName不行是因为，限定的超类很多，getName返回类型不可知，无法知道确切的返回类型，但是可以使用Object作为返回类型，因为Object是所有类的超类。

# 集合

## List

List当中有三个主要实现类，Vector、ArrayList、以及LinkedList。ArrayList与Vector一样，内部是通过Object数组来存储数据，Vector线程安全，而ArrayList是非线程安全，故ArrayList的效率会高些。数据超过数组的存储范围，那么就需要重新分配一个更大的数组，调用的是Arrays.copyOf方法。

对于LinkedList，底层是使用双向链表来进行实现的，LinkedList适合于插入于删除，而ArrayList适合于查找于修改。

## Set

HashSet底层是使用HashMap作为存储，



可以看到，map中键是可以保证唯一的，这样就满足了set中值唯一的要求了。HashSet可以进行遍历，但是遍历的顺序并不是插入的顺序，很显然，hash散列后就无序了。Hashset允许插入null。

LinkedHashSet继承自HashSet，区别在于LinkedHashSet当中维护着一个双向链表，用于保存元素插入时候的顺序。

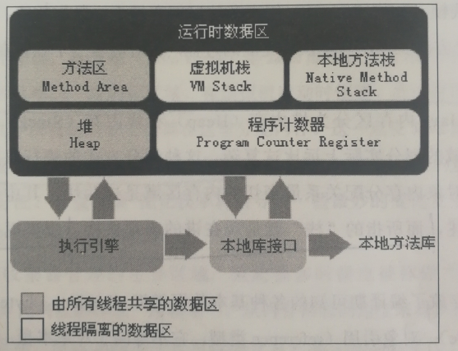
对于TreeSet，其底层也是使用的TreeMap，TreeSet可以保证当中元素是有序的，添加进去的值必须实现了comparable。

## Map

HashMap里面维持了一个table，对元素进行散列，如果说，散列到的位置没有元素，就直接插入，如果散列到的位置有元素，判断是否相同，相同就替换value，不相同就需要判断是否是树节点，如果是，那么就寻找当中是否有相同的，有就替换，没有就插入。如果不是树，那么就是一个链表，挨个遍历链表中的值，如果相同，那就替换，否则就插入，如果链表中的值超过8个，那么就将其生成为一颗树。

LinkedHashMap和TreeMap与上述的set相同，。

# Java内存区域



如图所示，Java的内存区可以分为上述的五个区域，

* 程序计数器。可以看作是当前线程所执行的字节码行号指示器。字节码解释器工作的时候就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支等等。程序计数器是线程私有的内存。
* Java虚拟机栈。Java虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的时候都会创建一个栈帧，这个栈帧用于存储局部变量表，操作数栈、动态链接、方法出口信息等，也就是说，每个方法从开始执行到执行结束，就是一个出栈跟入栈的过程。很显然，虚拟机栈也是一个线程私有内存。
* 本地方法栈。本地方法栈与虚拟机栈类似，只不过本地方法栈是为虚拟机使用到的Native方法服务。显然这个也是一个线程私有内存。
* Java堆。堆的唯一目的是为了存放对象实例，几乎所有的对象实例都在这里分配了。不过随着JIT编译器的发展与逃逸分析技术的逐渐成熟，堆上的分配也不是那么绝对了，可以在栈上分配空间了。
* 方法区。方法区与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。运行常量池是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。
* 直接内存。直接内存并不是虚拟机运行时数据的一部分，但是如果使用到了NIO，那么就需要调用Native数据库直接分配堆外内存。

对于对象的创建，基本步骤如下：

* 首先，去常量池中查看是否有该类的符号引用，并且检查这个符号代表的的类是否被加载、解析、和初始化过。如果没有，拿必须执行相应的类加载过程
* 类加载检查通过后，接下来虚拟机会为新生对象分配内存，对象所需要的内存大小在类加载完后便可以完全确定。有一个需要考虑的是，分配内存的时候会出现线程不安全，一种方法是使用同步处理，另一种方法是把内存分配动作按照线程，在不同的空间当中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲，等这一小块内存使用完成后，才需要使用同步在堆上。
* 在内存分配完成之后，虚拟机需要进行必要的初始化，会把内存空间都初始化为零值。
* 接下来，处理器需要对对象进行必要的设置。

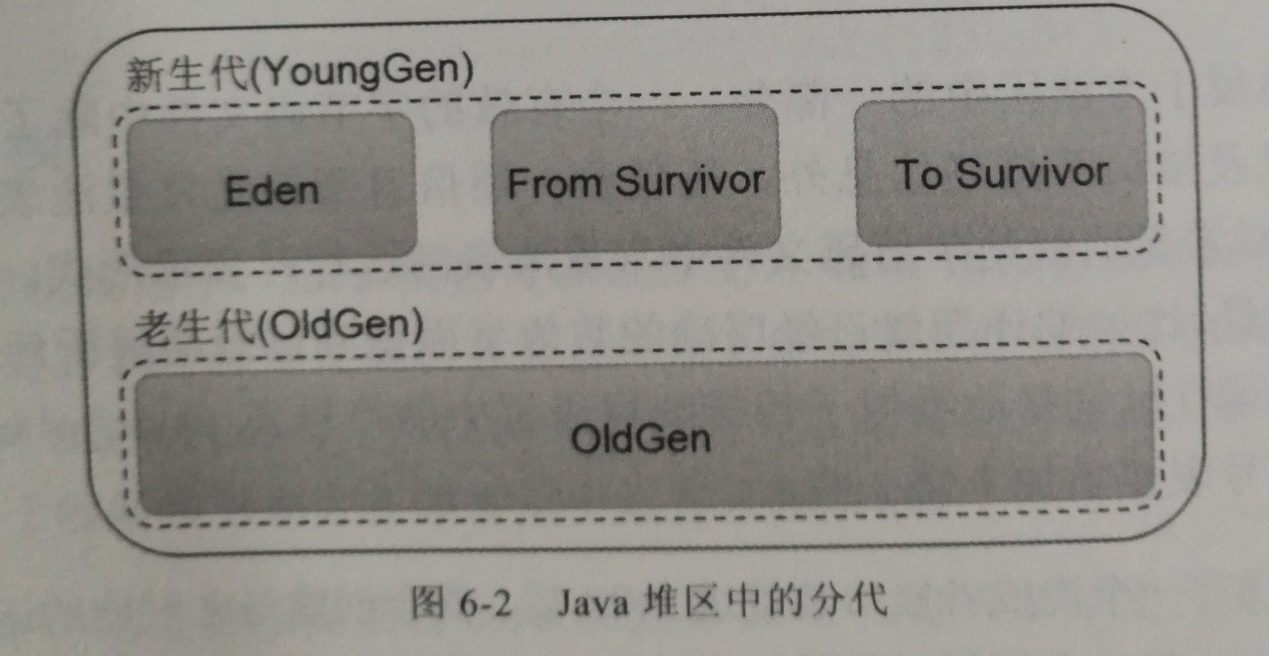
# 垃圾收集器与内存分配策略

引用计数算法，其实并不是Java主流的管理内存的方法，引用计数算法存在对象之间相互循环引用的问题。故主流的虚拟机不会使用该算法。 主流的虚拟机中，会使用可达性分析算法，来判断对象是否存活。这个算法的基本思路是通过一系列的称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链。如果说一个对象没有“GC Roots”可以到达，那么就可以说该对象是不可用的。一般来说，可以作为“GC Roots”对象包括这几种：虚拟机栈（栈帧中的本地变量表）中引用的对象、方法区中类静态属性引用的对象、方法区中常量引用的对象、本地方法栈中JNI引用的对象。

当我们标记出需要清理的对象的时候，那么我们就需要用到垃圾手机算法。

第一种是“标记-清除算法”。首先将需要回收的对象，在标记完成后统一回收所有被标记的对象。该算法主要存在两个不足，第一个就是效率的问题，标记与清除的效率都不高；另一个是空间的问题，清除后会存在很多不连续的内存碎片。

第二种是复制算法。复制算法将内存按容量划分为大小相等的两块，每次只使用其中的一块，当一块用完的时候，将存活着的对象复制到另外一块上，然后将使用过的内存空间一次清理掉。这种算法效率比较高，但是内存使用会缩小一半。一般来说，现在的虚拟机都采用这种收集算法来回收新生代，将内存划分为一个较大的Eden空间和两块较小的Sruvivor空间。堆区中的分代图如下。



复制的思想是在新生代中执行gc的时候，就需要将Eden和其中一个Survivor的垃圾进行标记，并清除，从而将剩余存活的对象移至另一个Survivor中。一般来说，Eden跟Survivor大小的比例为8：1，也就是说，只有10%内存被浪费掉了。当然，并不是每次清理后Survivor都够存活的对象来进行存储的。当存活对象在Survivor当中没有位置的时候，就需要向堆区的老年代进行借贷。

第三种就是标记-整理算法。标记-整理一般用于老年代。与标记-清除类似，标记的过程两者相同，但是后序的步骤不是直接清理，而是让所有的存活对象像一端移动，然后直接清理掉端边界以外的内存。

# 并发

## 线程池

线程池，是指管理一组同构同坐线程的资源池。线程池可以通过重用线程，而不是创建新的线程来，当用请求到达的时候，由于有工作线程的存在，那么就不需要等待，直接可以工作。线程池容量有基本容量，最大容量之分，基本大小就是线程池的目标容量，即在没有任务执行的情况下线程池的容量，并且只有在工作队列满了的情况下才会创建出超过这个数量的线程。最大容量表示可以同时活动的线程数量的上限。

线程池的创建可以通过Executors的静态工厂方法来创建。

* newFixedThreadPool，创建一个固定长度的线程池，每当提交一个任务时就创建一个线程，直到达到线程池的最大数量，这时，线程池的规模将不再变化。
* newCachedThreadPool，可以创建一个可缓存的线程池，如果线程池当前的规模超过了处理需求时，那么将回收空闲的线程，而当需求增加的时候，则可以添加新的线程，线程池的规模不受任何限制。
* newSingleThreadExecutor，这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个工作线程异常结束，会创建另一个线程来代替，newSingleThreadExecutor能确保依照任务在队列中的顺序来串行执行。
* newScheduledThreadPool，创建一个固定长度的线程池，而且以延迟或者定时的方式来执行任务。

Executor接口中只有一个execute方法，并没有能够管控Executor的生命周期的方法，为了管控生命周期需要使用ExecutorService，ExecutorService继承自Executor，它的生命周期有3种状态：运行，关闭，已终止。

* Shutdown方法将执行平缓的关闭过程，不再接收新的任务，同时等待已经提交的任务完成—包括那些还未开始执行的任务。
* ShutdownNow是一种粗暴的关闭过程，它将尝试取消所有运行中的任务，并且不再启动队列中尚未执行的任务。

使用Executor线程的时候，需要注意的是避免线程饥饿死锁的情况：例如现在使用的是

单线程的Executor，并且已经有一个任务A在执行，而该任务在执行的过程中还向Executor中提交了一个任务B，并且任务A需要使用提交任务B运行后的值，这样任务A就在等待任务B执行，由于是单线程的Executor，任务B需要等任务A运行完成后进行运行，这样就存在互相等待而死锁的情况。这是由于任务A需要依赖于任务B的运行结果，在线程池不够大的时候，会出现这样的饥饿死锁。

当线程池中的工作线程使用完，再提交的任务就会进入任务列队。在线程池中，这些请求会在一个由Executor管理的Runnable队列中等待，而不是创建线程来争抢CPU资源，ThreadPoolExecutor使用的是BlockingQueue来保存等待执行的内容，



基本任务的排队方法也有3种：无界队列、有界队列和同步移交。newFixedThreadPool和newSingleThreadExecutor在默认的情况下使用的是无界的LinkedBlockingQueue。如果工作者线程都处于忙碌，那么任务将在队列种等待。更为稳妥的资源管理策略是使用有界队列，但是当有界队列也填充满了，就需要使用到饱和策略了。对于非常大的或者无界的线程池，可以通过使用SynchronousQueue来避免任务排队，如果要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接受这个元素，newCachedThreadPool工厂方法就是使用了SynchronousQueue。

饱和策略有四种：中止、抛弃、抛弃最旧、调用者运行。

* 中止：中止策略是默认的饱和策略，该策略将抛出未检查的RejectedExecutionExeception。调用者可以捕获这个异常，然后根据需求编写自己的代码。
* 抛弃：当新提交的任务无法保存到队列中等待执行时，抛弃策略将悄悄的抛弃该任务。
* 抛弃最旧：抛弃最旧策略会抛弃下一个会被执行的任务，然后重新尝试提交新的任务，如果队列是一个优先级队列，那么抛弃最旧策略将会抛弃下一个将会抛弃优先级最高的任务，这两者不要结合。
* 调用者运行：调用者运行策略是实现了一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将任务回退给调用者，它不会在线程池的某个线程中执行新的任务，而是在调用了execute的线程里面执行该任务。

# NIO

NIO相对于传统的IO的优势体现在非阻塞，

