目录

[线程： 2](#_Toc26439)

[equals和==区别： 4](#_Toc28411)

[重写equals方法时，为什么要必须重写hashCode()? 4](#_Toc16377)

[Java中类的加载过程 6](#_Toc19875)

[1>加载 6](#_Toc14314)

[2>验证 7](#_Toc16308)

[3>准备 7](#_Toc4903)

[4>解析 7](#_Toc23628)

[5>初始化 8](#_Toc27279)

[6>使用、卸载 8](#_Toc9761)

[Java中类的加载器（1.8） 8](#_Toc5646)

[Java中类的加载机制之双亲委派模型 9](#_Toc7502)

[Java内存分配 10](#_Toc1449)

[集合 16](#_Toc2229)

[守护线程： 22](#_Toc26595)

[锁： 22](#_Toc31573)

**线程：**

无界队列会导致线程积压，消耗内存。

newFixedThreadPool:

无界队列

newCacheThreadPool、

newSingelThreadPool、

只有一个线程

newScheduledThreadPool

new ThreadPoolExcutor

线程池的提交优先级、执行优先级

提交优先级:核心线程>队列>非核心线程

执行优先级:核心线程>非核心线程>队列

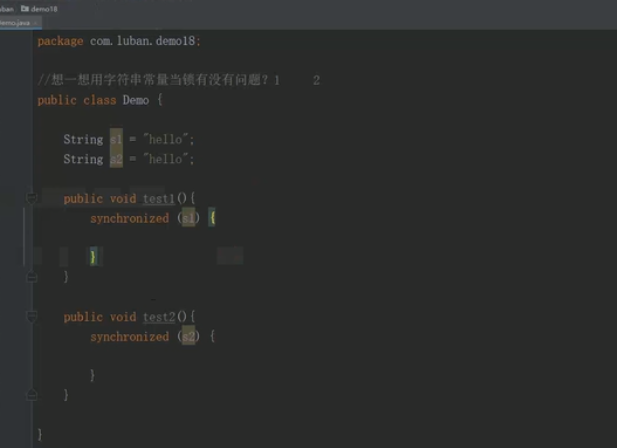
JMM模型(java线程内存模型)

可见性：线程之间的可见性，一个线程修改的状态对另一个线程是可见的，即一个线程修改的结果，另外一个线程马上就能看到。Java中volatile、synchronized 和 final 具有可见性。

原子性：

有序性：

不要用常量做同步,因为可能第三方包用了相同的常量做同步，这样会导致一直获取不到锁：



图中，s1和s2指向的是同一个地址，所以test1和test2方法其实是获取的同一个对象的锁。

wait和notify、notifyAll

都是Object基础类的方法，而非Thread类的方法。当需要调用以上方法时，需要对竞争资源加锁(即需要将这些方法放到synchronized(obj)中)。

wait会释放锁,但是notify后不会立即释放锁，直到执行完synchronized。

线程优先级：

线程优先级有10个等级，分别用整数1-10表示。其中1位最低优先级，10为最高优先级，5为默认值。但是，线程的优先级与代码执行顺序之间并无关联，只不过,优先级高的线程获取CPU资源的概率较大，优先级低的并非没机会执行。线程的优先级具有继承性，比如A线程启动B线程，则A和B的线程优先级是一样的。

sleep()与wait():

sleep()是Thread的静态方法。

被调用后，都会暂停当前线程并让出cpu的执行时间，但是sleep不会释放当前持有的对象锁资源，到时间后会继续执行(不是立即执行，只是达到可执行的状态，需要获取到cpu资源后方可执行)；而wait被调用后立即释放锁资源，需要调用对象的notify()/notifyAll()方法唤醒.

volatile()：

Java之八大基础数据类型

byte 8位 -128~127

short 16位 -216-1~216-1-1

int 32位 -232-1~232-1-1

long 64位 -264-1~264-1-1

float 32位

double 64位

char 16位

boolean true或false

类型转换：

char--> 自动转换：byte-->short-->int-->long-->float-->double

**equals和==区别：**

1. equals是方法，==是运算符;
2. 如果使用==比较基础数据类型，则是比较两者数值是否相等，如果比较的是引用数据类型，则比较两者地址是否相等;equals是用来比较两个对象的内容是否一致;
3. 如果equals和==都用来比较对象时，且两个对象的引用地址一致时，equals可能返回true或者false，这要取决于是否重写了对象的equals方法已经重写的内容，而==一定返回true。

**重写equals方法时，为什么要必须重写hashCode()?**

hashCode()方法是用来获取对象的哈希码值，这个值是用来确定该对象再哈希表中索引所在的位置。

equals()是用来判断两个对象是否相等，如果对象没有重写equals方法，即比较两个对象的地址是否相同，等价于”==”。

先看看代码的例子,有两个Person类，Person2只重写了equals()，Person1重写了equals()和hashCode():





Main():



运行结果：



根据上图的结果想要说明的是:

①如果只重写equals方法，虽然equals的结果是true，但是如果把对象放到HashSet或者其他自动去重的集合中，因为没有重写hashCode，所以此时两个对象的hashcode值其实是不一样的，HashSet会把他们当作两个不同的对象，不会去重。所以，重写equals时必须重写hashCode,是指在需要用到集合(HashSet)去重对象时，必须重写hashCode().

②如果两个对象equals相等，那么他们的hashCode值一定相等。

③两个对象如果hashCode值一样，他们也不一定相等，需要调用equals来比较。

④hashCode()默认是在堆栈上的对象产生独特的哈希值。如果没有重写hashCode()，那么两个对象无论如何都不会相等。

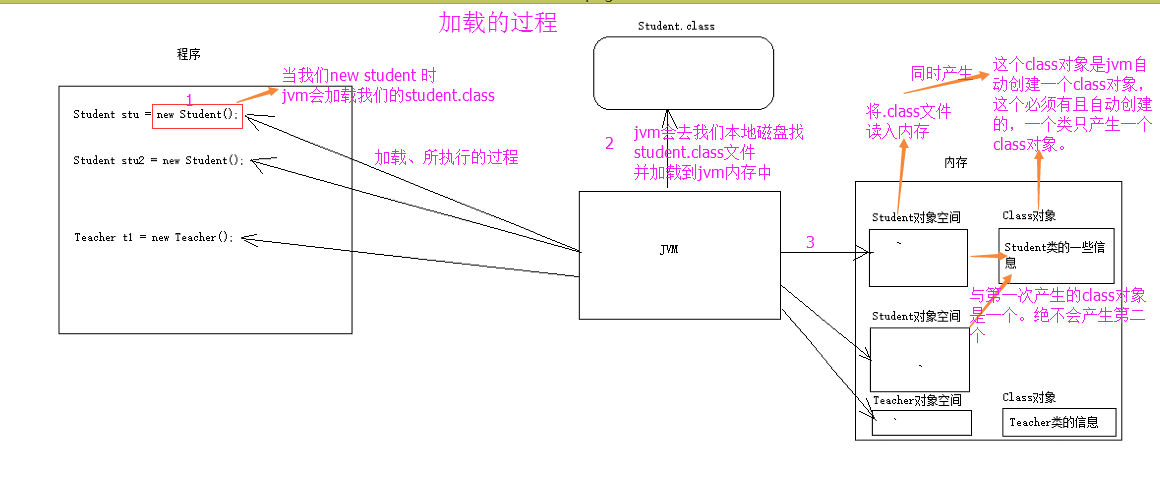
**Java中类的加载过程**

.java文件在编译成.class文件之后生命周期会经历以下7个阶段，前面5个阶段是类的加载过程：

1>加载

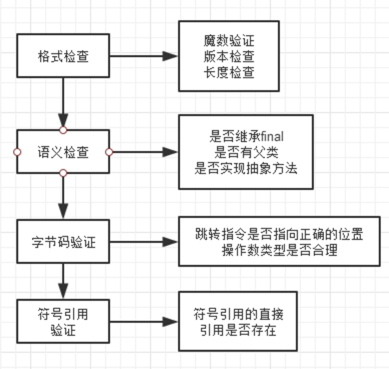
由类的加载器根据类的全限定名读取此类的二进制字节流到jvm中，并存储在运行时数据区的方法区内，然后将其转换为对应的class对象，该对象描述了这个类创建的对象的所有信息，比如成员变量、构造方法、成员方法等。

类被加载到方法区后主要包含 常量池、类型信息、字段信息、方法信息、类加载器的引用、对应class实例的引用等信息。对应class的实力对象会放到堆(Heap)中，作为开发人员访问方法区中类定义的入口和切入点



2>验证

该阶段只要是确保.class文件的字节流满足java虚拟机规范，不会造成安全错误。



3>准备

为类中的所有静态变量分配内存，并为其设置一个默认初始值，如果是被final修饰的静态变量，则直接赋予原值。

4>解析

将二进制字节流中的符号引用转为直接引用。

说明：

符号引用，即一个字符串，但是这个字符串给出了一些能够唯一性标识的一个方法或者一个变量或者是一个类的相关信息。

直接引用，可以理解为一个内存地址，或者一个偏移量。比如类方法、类变量的直接引用就是指向方法区的指针；而实例方法，实例变量的直接引用则是从实例的头指针开始算起到这个实例变量位置的偏移量。

举个例子来说，现在调用方法hello(),这个方法的地址是0xaabbccdd，那么hello()就是符号引用，0xaabbccdd就是直接引用。

5>初始化

为标记为常量的属性赋值的过程。即对static修饰的变量或者代码块进行初始化，如果初始化的时候，其父类未初始化，则优先初始化其父类。

6>使用、卸载

初始化完成后，就可以对该类使用了，使用完之后，在方法区垃圾回收的过程中进行卸载(垃圾回收);

**Java中类的加载器（1.8）**

类加载器的任务就是根据类的全限定名来读取此类的二进制字节流到JVM中，然后转换成一个与此类对应的java.lang.Class对象实例，这个对象实例用来表示一个Java类。

所有的类加载器都继承了抽象类java.lang.ClassLoader。

Java中的类加载器大致可以分成2类，一类是系统提供的，另一类是自建的。系统提供的主要有三个：

BootstrapClassLoader:启动类加载器，用来加载jre\lib目录中的特定名称的文件(java的核心类库)，如rt.jar、charsets.jar等,而不是该目录的所有文件。，他不继承java.lang.ClassLoader;

ExtClassLoader:扩展类加载器，负责加载jre绝对路径\lib\ext目录中或者的文件

AppClassLoader:应用加载器，负责加载用户类路径中的文件；

用户可以直接使用扩展类加载器或者应用加载器来加载自己的类，但是无法使用启动类加载器加载自己的类。

CustomClassLoader：自定义的加载器，用户自定义的类加载器,可加载指定路径的class文件

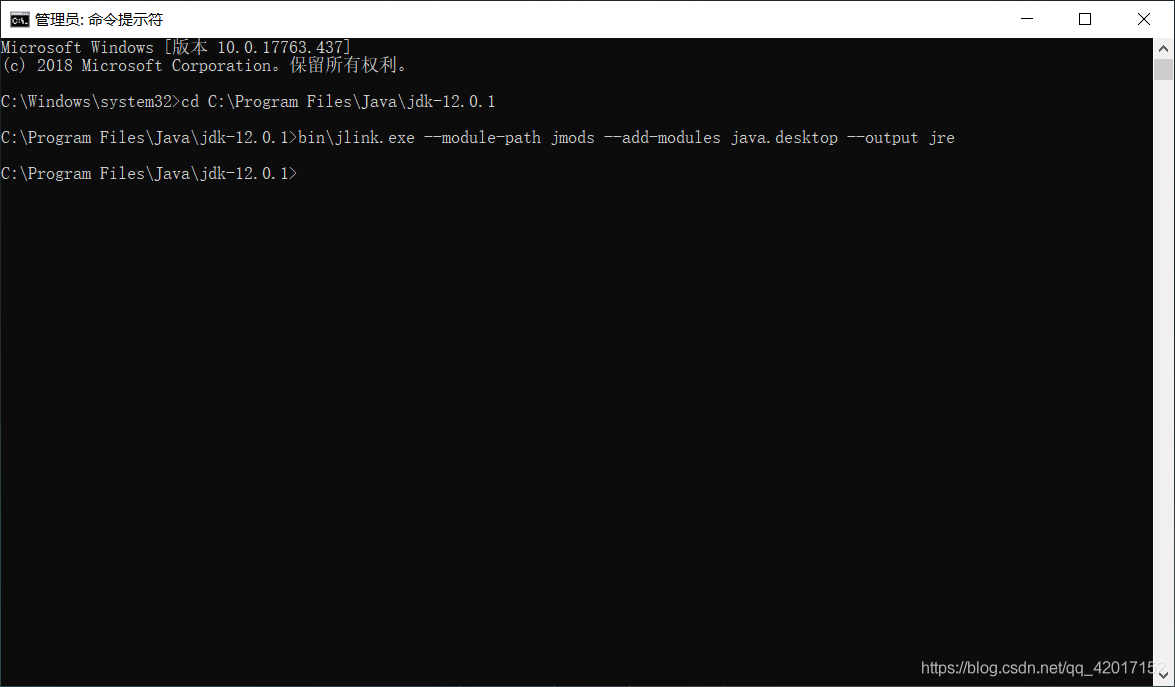
为什么jdk11安装之后没有jre目录？如何生成jre目录?

随着虚拟化、容器化、微服务化的普及和流行，庞大的jre跟不上时代的发展了：作为一个运行时环境，里面包含了太多不需要的东西。比如，如果一台主机上的jre仅仅是用来运行Apache tomcat,那么jre中用来支持SWT的部分肯定是不需要的，既浪费空间又降低效率.

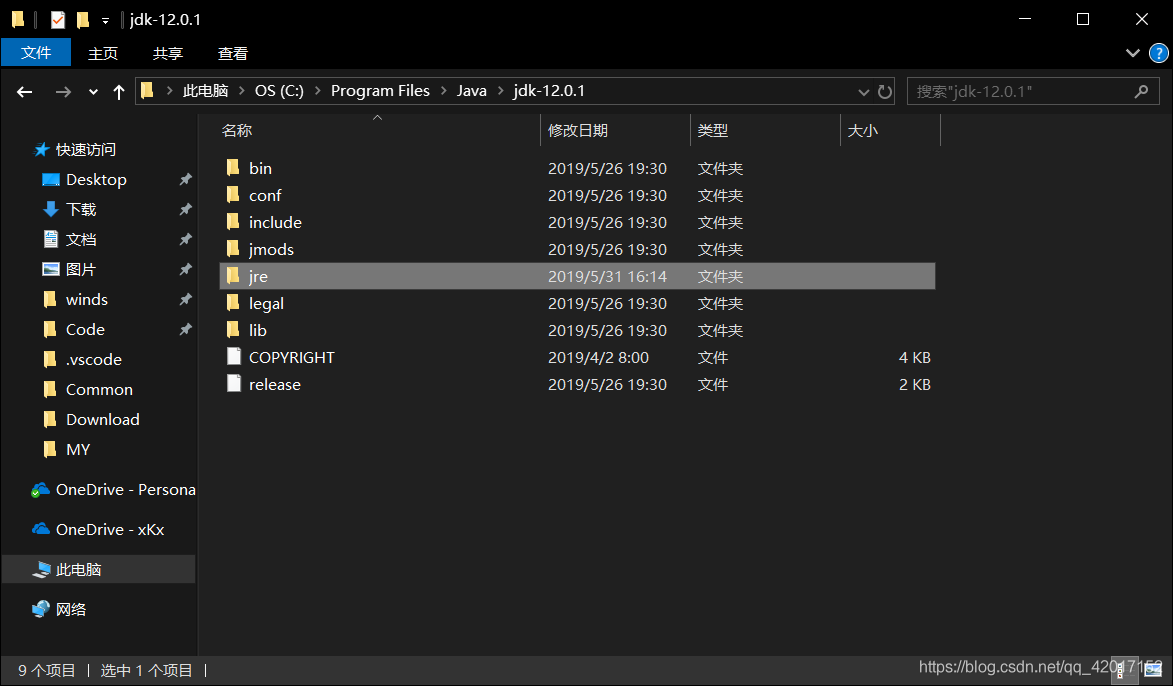
Jkd从1.9之后开始引入module system,使得开发者能够按照自己的应用创建一个最小的运行时环境--一个微服务应用的部署仅仅需要一个非常小的runtime(比如仅仅20M),而不是像以前都需要一个上百兆的jre作为运行时环境。

如何生成jre目录呢？打开cmd，进入jdk11安装目录之后，执行以下命令:

bin\jlink.exe --module-path jmods --add-modules java.desktop --output jre



执行完之后：



**Java中类的加载机制之双亲委派模型**

类的加载方式分为隐式加载和显示加载。隐式加载值得时程序在使用new等方式创建对象时，会隐式调用类的加载器把对应的类加载到JVM中。显示加载指的是通过直接调用class.forName()方法来把需要的类加载到JVM中。

当程序启动时，只把需要的类加载到JVM中，其他的类只有被使用到的时候才会加载到JVM中，这种方式一方面可以加快类加载速度，另一方面可以减少运行时对内存的开销。

双亲委派概念：

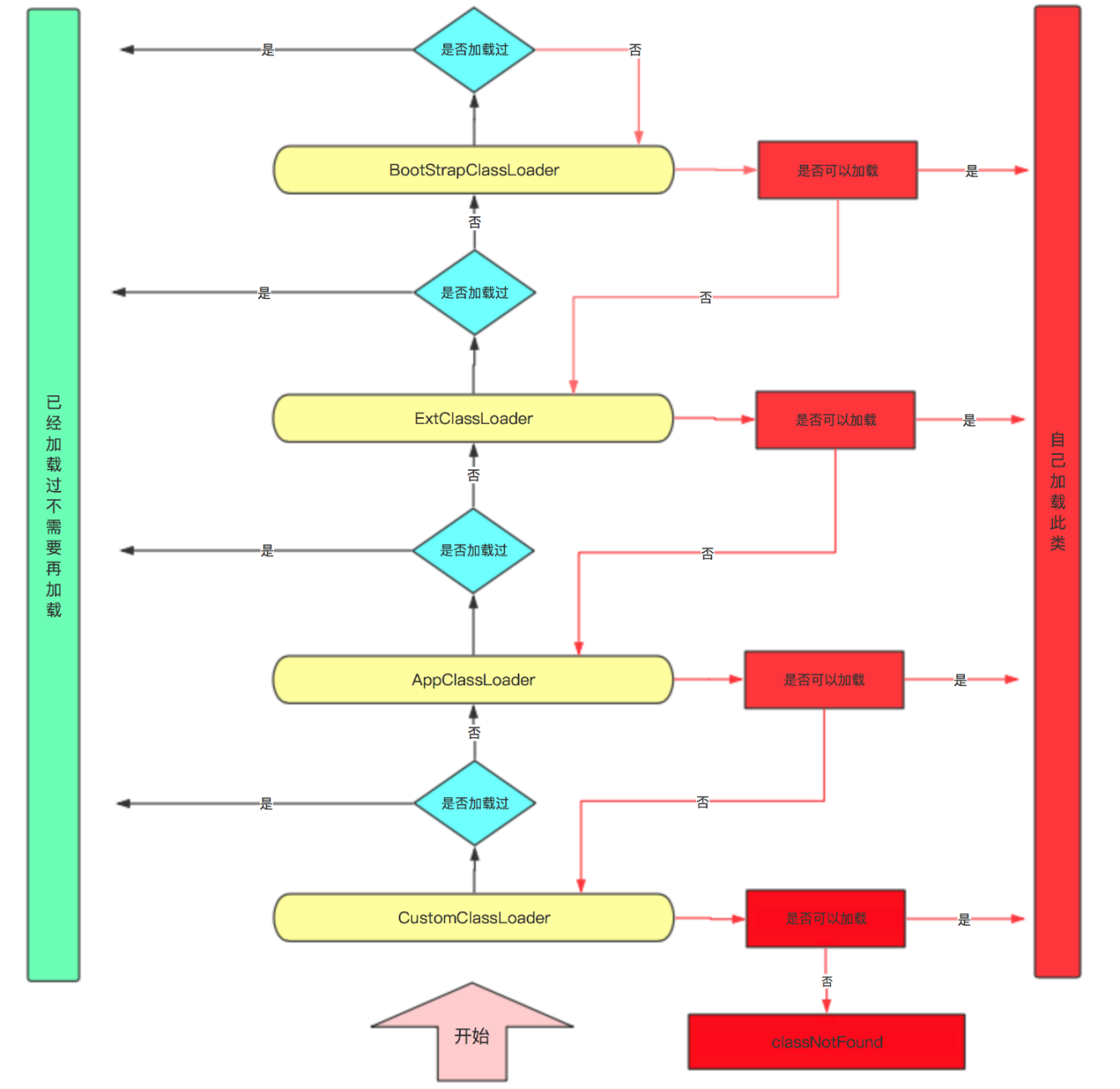
当某个类加载器需要加载某个.class文件时，它首先把这个任务委托给他的上级类加载器，递归这个操作，如果上级的类加载器没有加载，自己才会去加载这个类。

类加载器加载的过程：

1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接 返回。

2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加 载器加载（即调用parent.loadClass(name, false);）.或者是调用bootstrap类加载器来加 载。

3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的 findClass方法来完成类加载。



双亲委派模型的作用：

1. 防止.class文件重复加载；
2. 保证核心class文件不能被篡改。

**Java内存分配**

寄存器：我们无法控制

静态域：static定义的静态成员

常量池：编译时被确定并保存在.class文件的final常量值和一些文本修饰的符号引用。

非RAM存储：硬盘等永久存储空间。

堆内存：new创建的对象和数组，由java虚拟机自动垃圾回收期管理，存取速度慢

栈内存：基本类型的变量和对象的引用变量(堆内存空间的访问地址)，速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

**Class.forName()与ClassLoader.loadclass()区别**

Class.forName得到的是已经初始化完成的类，而ClassLoader.loadclass()得到的是还没有链接(验证、准备、解析)的类。

**Outofmemory与stackoverflow**

两者都是内存溢出,区别是:

Stackoverflow栈内存溢出，是线程运行时报的错，表示当前线程使用的占内存已经超过最大值了。一般是由于递归调用，或者申请的局部变量太大，产生了超过占内存最大值的数据。增大栈内存，也会出现栈内存溢出，只是时间推迟了。

Outofmemory堆内存溢出，是数据创建前报的错，表示当前剩余的内存已经不够了，不能创建新的数据了。

**内存溢出与内存泄漏**

内存溢出：Outofmemory，是指系统不能再分配出程序执行所需要的空间，于是产生溢出。

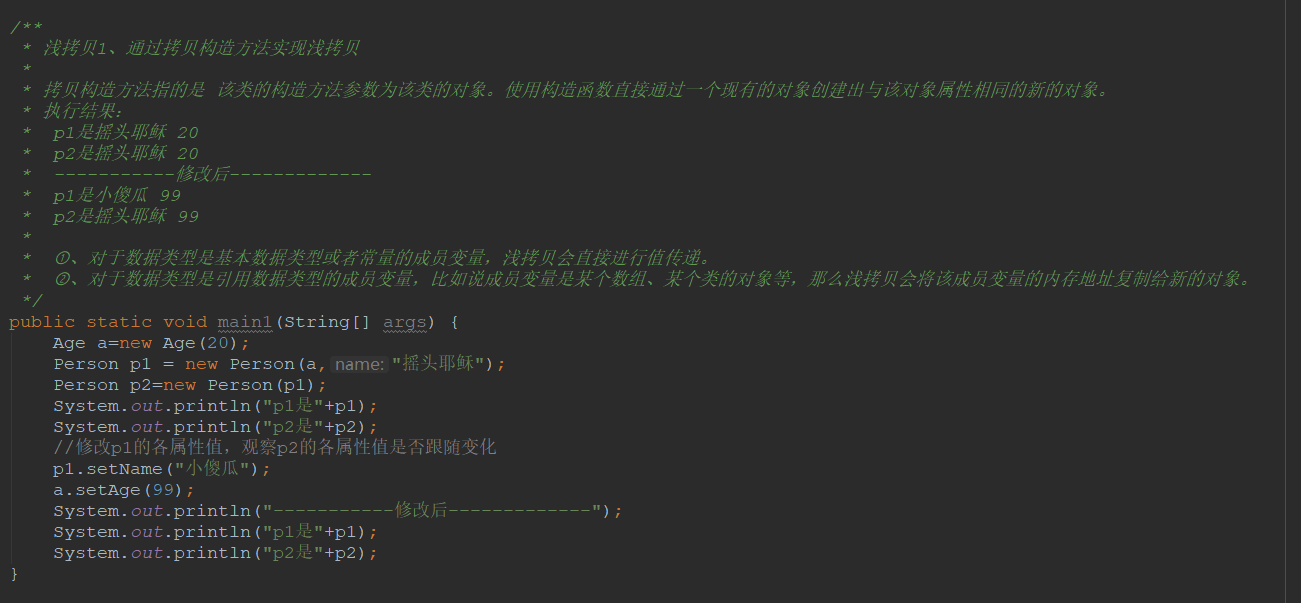
内存泄漏：memory leak,是指向系统申请分配内存并使用完之后却无法释放。一次泄漏危害可以忽略，大内存泄漏堆积后会导致内存溢出。内存泄漏可以分为以下4类：

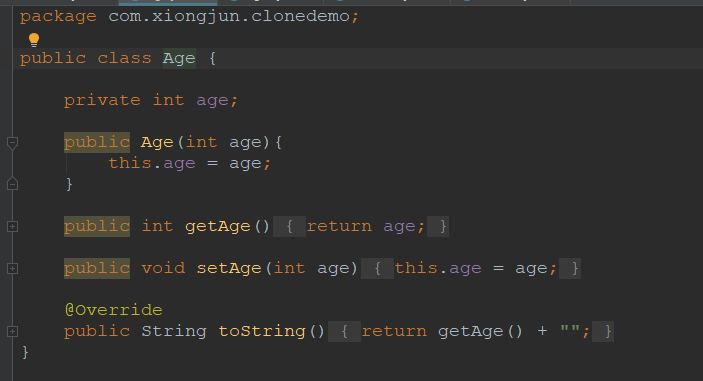
1. 常发性内存泄漏，发生内存泄漏的代码会被多次执行到，每次执行都会导致一块内存泄漏。
2. 偶发性内存泄漏，在特定环境下才会发生。
3. 一次性内存泄漏，发生内存泄漏的代码只会被执行一次。
4. 隐式内存泄漏，程序在运行过程中不停的分配内存，但是知道程序结束的时候才释放内存。这种不及时释放内存也可能导致最终好景系统的所有内存。

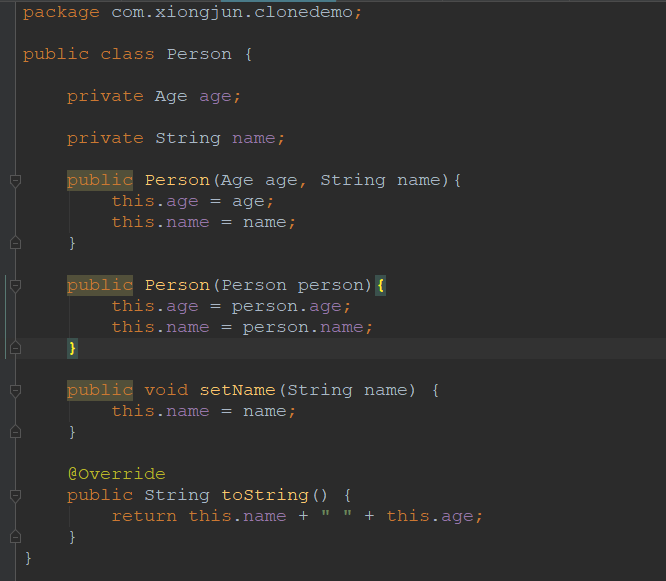
**浅拷贝与深拷贝**

浅拷贝：

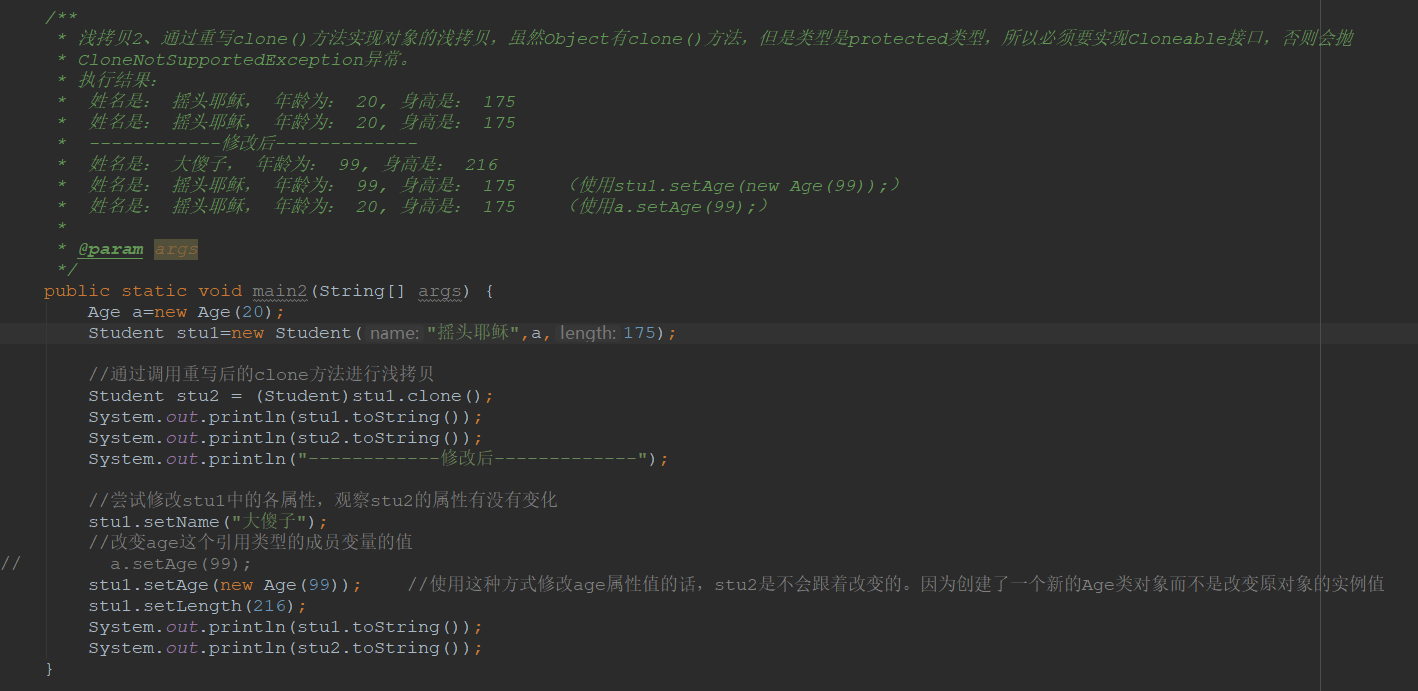
方法一：

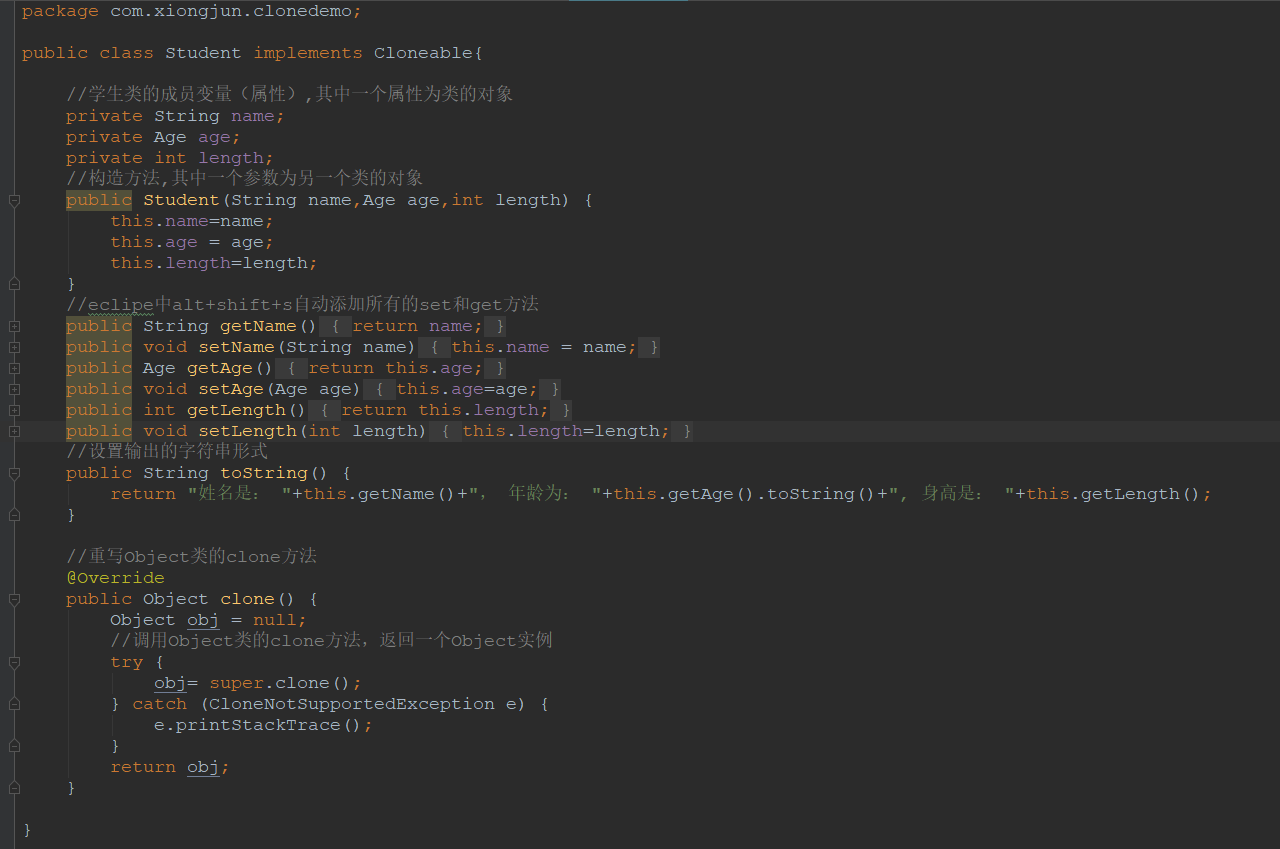






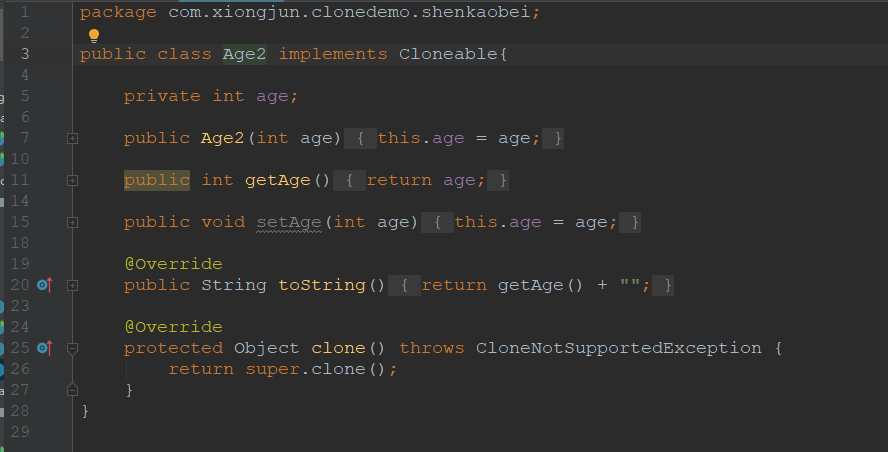
方法二：

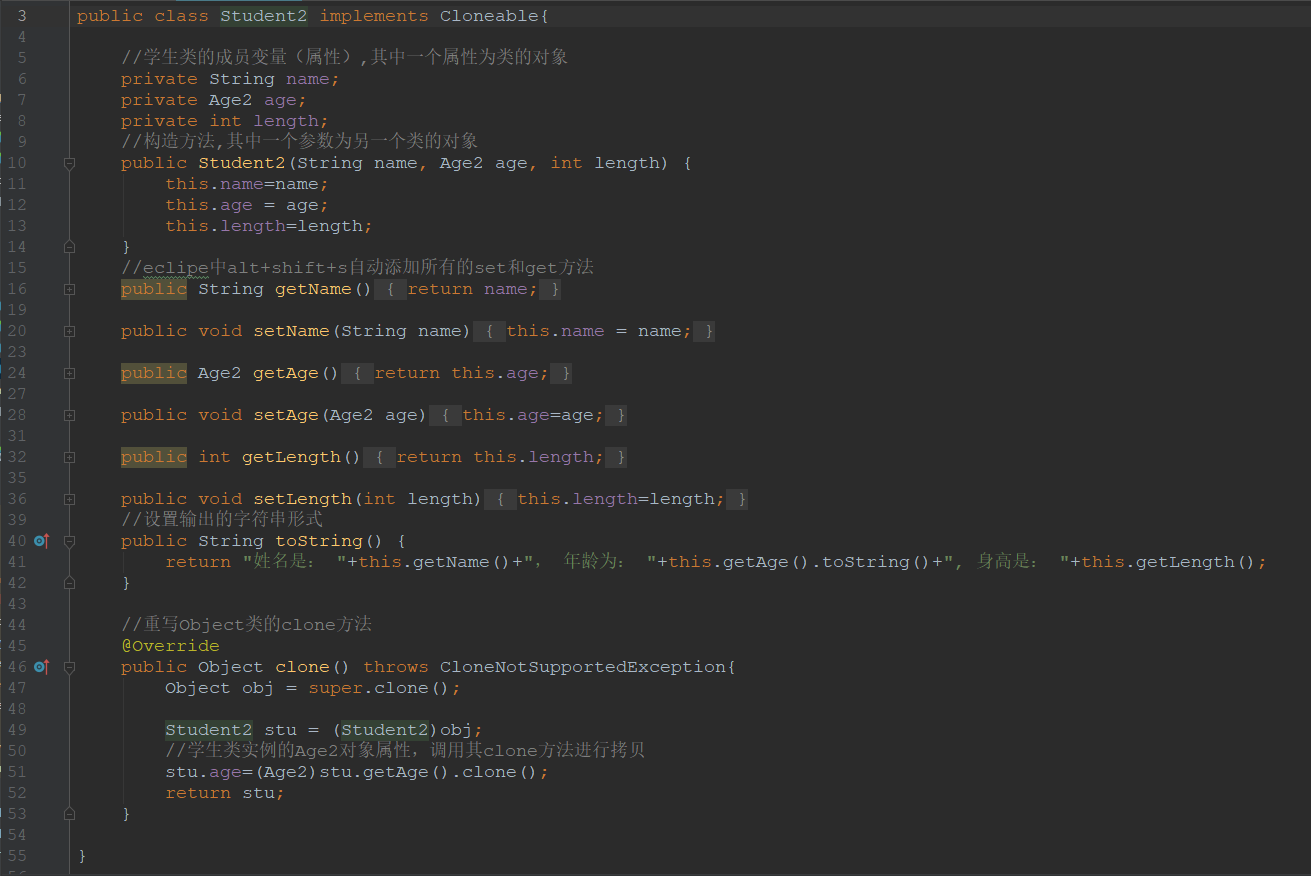




深拷贝1：

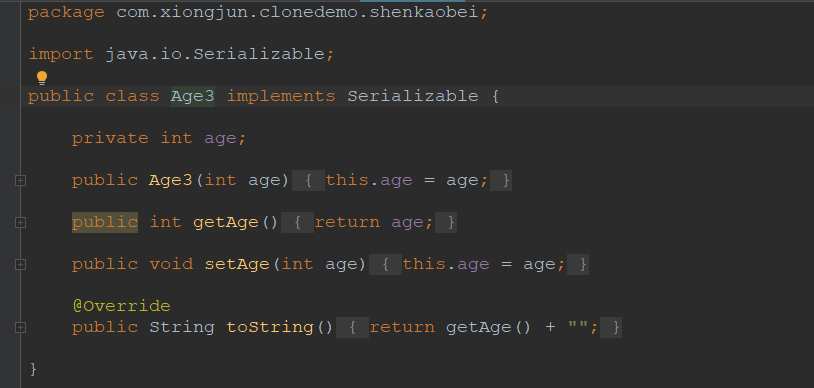


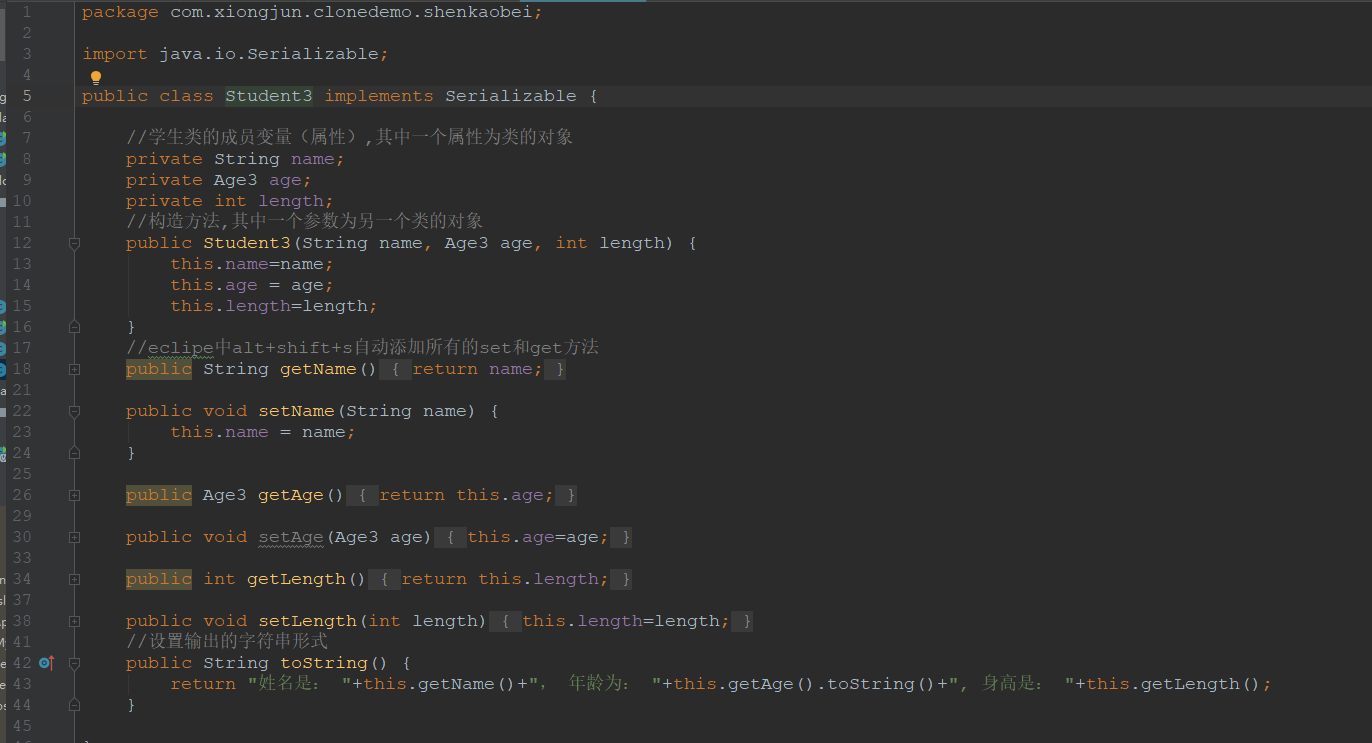




深拷贝2：







**OSGI（open service gateway initiative）**

他是与与双亲委派模型不同的一中机制，实现了在运行部署系统之后还能再不停止服务器的情况下直接把某些模块拿出来，其他模块的功能也不受影响，即模块化解耦。

OSGI技术里面内嵌了一个jetty服务器，

**增加Eden区，导致Minor GC时间间隔变长，会不会导致MinorGC的执行时间变长？**

单次Mainor GC的时间是由两部分组成：T1(扫描新生代)和T2(复制存活对象)，因为通过在虚拟机中，复制对象的成本要远高于扫描对象的成本，而增加Eden区增加了T1的时间，所以在这个时间过程中，大对象有可能已经变成了垃圾，所以，gc执行的时候直接清理了，减少了复制的过程，所以这样会节省Minor GC的执行时间。如果大对象没有变成垃圾，还处于存活状态，反而只会增加T1的时间，没有改变T2时间，所以此时MinorGC的时间会变长。

综上所述，单词Minor GC的时间更多取决于GC后存活对象的数量，而非Eden区的大小。

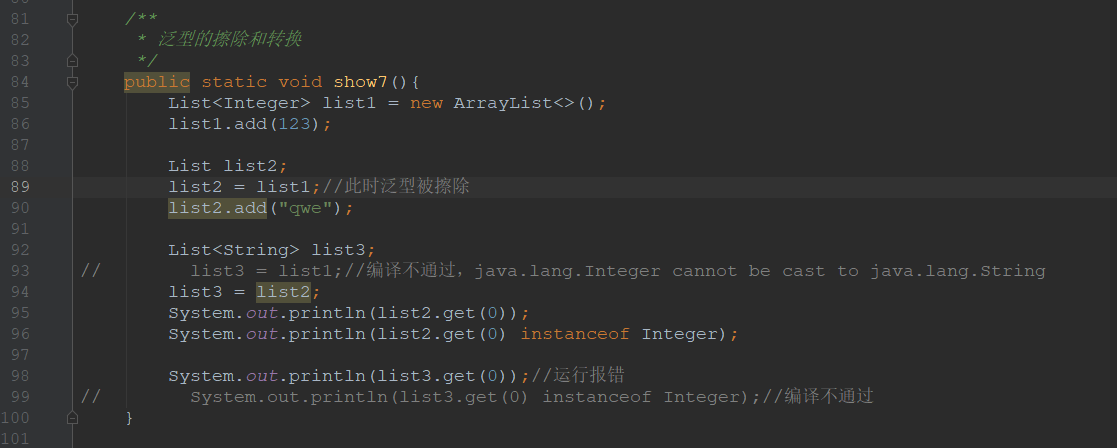
**List<? extends T>和List<? super T>**

List<? extends T>可以接受任何继承自 T 的类型的 List，但是不能添加元素，只能添加空元素；

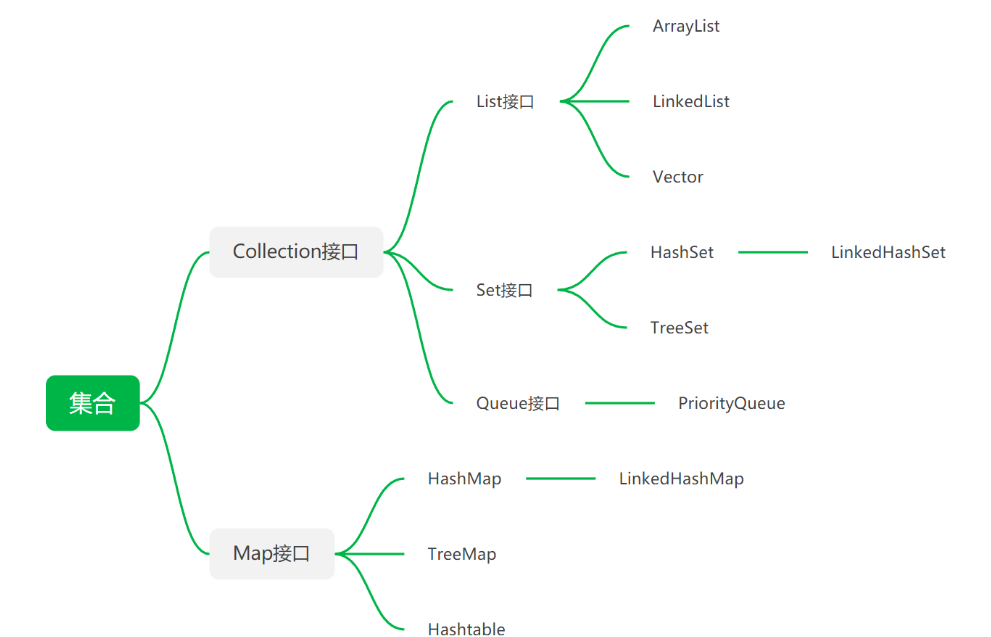
List<? super T>可以接受任何 T 的父类构成的 List，但是不能用for进行遍历。



**泛型的擦除和转换**



**集合**



|  |  |  |
| --- | --- | --- |
| ArrayList  (List) | ArrayList是基于动态数组实现的;  ArrayList的默认初始化容量是10，每次扩容时候增加原先容量的一半，也就是变为原来的1.5倍;  删除元素时不会减少容量，若希望减少容量则调用trimToSize();  随机查找速度快，插入删除速度慢（插入删除均需移动元素）;  非线程安全（需要同步时可使用Vector类）;  能存放null值;  如果想做成线程安全，可以用如下方式：  List<Object> arryList = Collections.*synchronizedList*(new ArrayList<>()); | +add();+add(index, obj);  +addAll();+addAll(index, obj);  +size();+toArray();转数组  +clear();+remove();+removeAll();  +get();+contains();  +set(index, obj);替换位置的元素  ... |
| LinkedList  (List) | 一种可以在任何位置进行高效地插入和删除操作的有序序列;  随机查找速度慢，插入删除速度快;  底层实现是链表（双向链表）; | +add();+add(index, obj);  +addAll();+addAll(index, obj);  +addFirst();+addLast();  +size();+toArray();转数组  +clear();+remove();  +removeAll();  +get();+contains();  +set(index, obj);替换位置的元素  ... |
| Vector  (List) | Vector底层也是数组，与ArrayList最大的区别就是：同步(线程安全)；  ArrayList在底层数组不够用时在原来的基础上扩展0.5倍，Vector是扩展1倍；  Vector的默认初始化容量是10；  能存放null值; | 方法跟ArrayList差不多 |
| HashSet  (Set) | 元素是无序不可重复的,可以存在null;  非线程安全;  像HashSet中插入对象时，HashSet调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 hashCode值来决定该对象在HashSet中存储位置。修改了元素，元素的hashCode值也会改变。它的实现是依赖HashMap。 |  |
| LinkedHashSet  (Set) | 有序(先进先出)，不可重复，可以存在null,非线程安全，是HashSet的子类，根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。  LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，因为它是根据链表维护元素的次序的，但是插入时性能稍微逊色于HashSet(百万数据量时才能体现出)。它的实现依赖于LinkedHashMap |  |
| TreeSet  (Set) | 元素是有序不可重复的,不能存在null;  非线程安全;  依赖TreeMap实现;  支持两种排序方式，自然排序和自定义排序。  自然排序:实现Comparable接口，重写了compareTo()来进行排序。  自定义排序:  实现Comparable接口或者Comparator接口，重写compareTo()方法。  如下图 | 注意：1、TreeSet只能添加相同类型的元素。  Set set = new TreeSet();  set.add(1);  set.add("2");  System.out.println(set);  这种是错误的；  2、获取TreeSet的元素时，元素的类型必须时实现了Comparable接口或者Comparator接口 |
| PriorityQueue  (Queue，优先队列) | 参考：  <https://www.cnblogs.com/wei-jing/p/10806236.html>  https://blog.csdn.net/u010623927/article/details/87179364  PriorityQueue是基于优先堆的一个无界队列，这个优先队列中的元素可以默认自然排序或者通过提供的Comparator（比较器）在队列实例化的时排序。  不允许null;不支持不可比较的对象(未实现Comparable或Comparator接口的对象);非线程安全;默认长度是11，利用二叉小顶锥实现(任意一个非叶子节点的权值，都不大于其左右子节点的权值），也就意味着可以通过数组来作为PriorityQueue的底层实现)。 | +peek()//返回队首元素  +poll()//返回队首元素，队首元素出队列  +add()//添加元素  +size()//返回队列元素个数  +isEmpty()//判断队列是否为空，为空返回true,不空返回false |
| PriorityBlockingQueue | 线程安全; |  |

TreeSet-自然排序：





执行结果：



TreeSet-自定义排序：







执行结果：

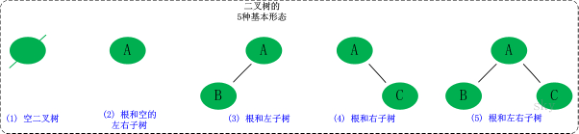


|  |  |  |
| --- | --- | --- |
| HashMap | key不能重复，但是value可以重复;  非线程安全;无序;key和value都允许为空; | 1.8以前是数组+链表  18及以后是数组+链表+红黑树 |
| TreeMap | 对键有序的遍历 | key 必须实现 Comparable 接口或者在构造 TreeMap 传入自定义的Comparator |
| LinkedHashMap | 按照插入的顺序存储;  非线程安全 |  |
| Hashtable | Key、value都不能为空  线程安全,实现线程安全的方式是在修改数据时锁住整个哈希表，效率低，ConcurrentHashMap做了相关优化;  Hashtable的初始size为11，扩容：newsize = olesize\*2+1。 | 数组+链表 |
| ConcurrentHashMap | 线程安全;，比Hashtable效率高(采用的是分段锁) |  |

**数据结构之树：**

什么是二叉树？

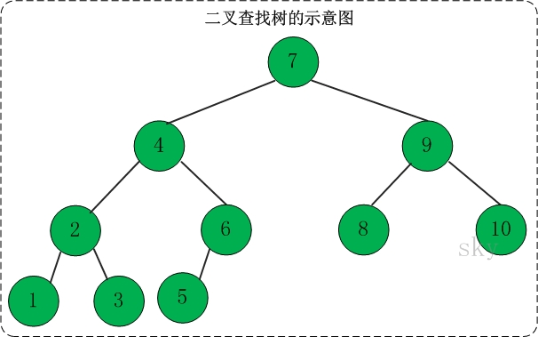
二叉树是每个节点最多有两个子树的树结构。他有5中基本形态：



1. 排序二叉树(二叉查找树\二叉搜索树)：

如果二叉树每个节点满足：左子树所有节点值小于它的根节点值，且右子树所有节点值

大于它的根节点值，则这样的二叉树就是排序二叉树。



在二叉查找树中：

(01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

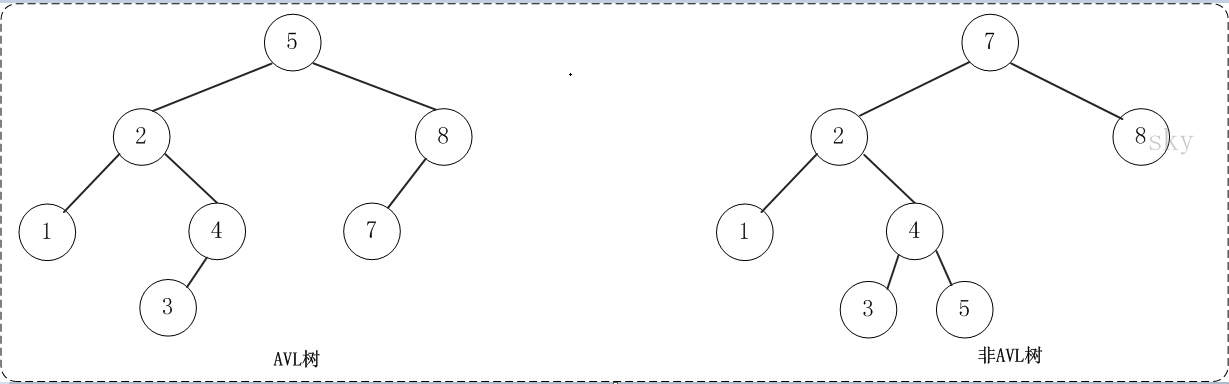
(02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

(03) 任意节点的左、右子树也分别为二叉查找树。

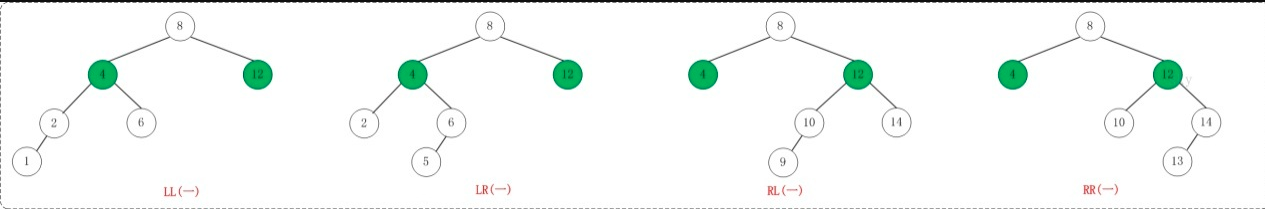
(04) 没有键值相等的节点（no duplicate nodes）。

1. 平衡二叉树(AVL树)：

平衡二叉树是高度平衡的查找二叉树，还满足任何节点左子树和右子树的高度之差的绝对值不超过1, 并且树中的每个节点的左子树和右子树都是AVL树。如下图，左边是AVL树，它的任何节点的两个子树的高度差<=1；右边的不是AVL树，其根节点的左子树高度为3，而右子树高度为1。



如果在AVL树中进行插入或者删除节点，可能会导致AVL树失去平衡(即某个节点的左子树和右子树的高度差>1)，失去平衡的二叉树会有四种情况，如下图：



LL：LeftLeft，也称“左左”。插入或删除一个节点后，根节点的左孩子（Left Child）的左孩子（Left Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

RR：RightRight，也称“右右”。插入或删除一个节点后，根节点的右孩子（Right Child）的右孩子（Right Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

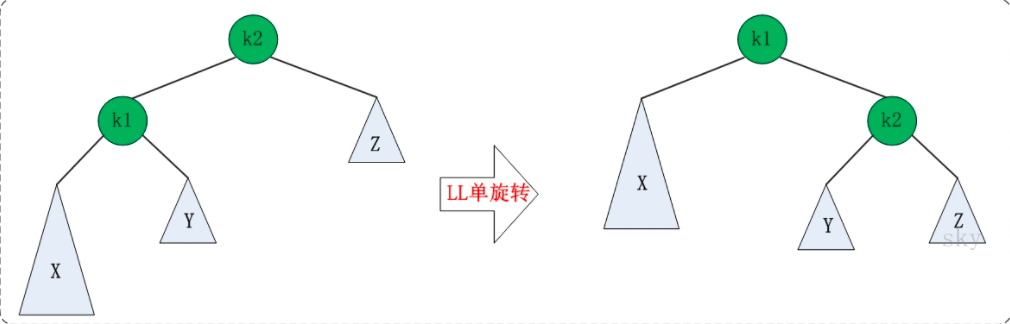
LR：LeftRight，也称“左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

RL：RightLeft，也称“右左”。插入或删除一个节点后，根节点的右孩子（Right Child）的左孩子（Left Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

AVL树失去平衡之后，可以通过旋转使其恢复平衡。下面分别介绍四种失去平衡的情况下对应的旋转方法。

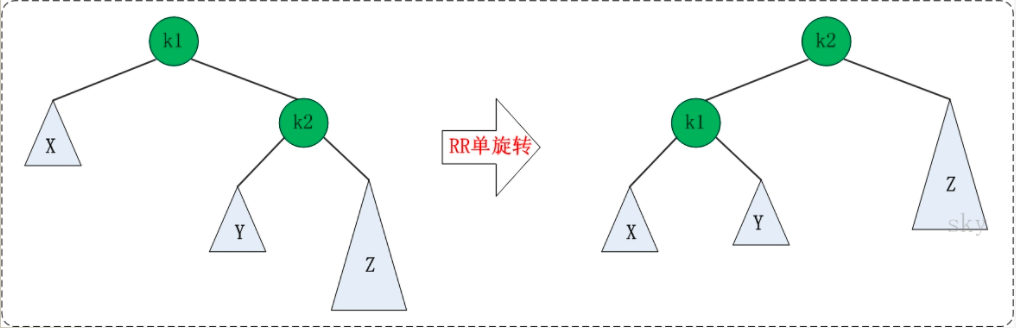
对于LL:可以通过一次旋转让整个树恢复平衡。步骤：

1. 将根节点的左子节点作为新的根节点；
2. 将新跟节点的右子节点作为原根节点的左子节点；
3. 将原根节点作为新根节点的右子节点。



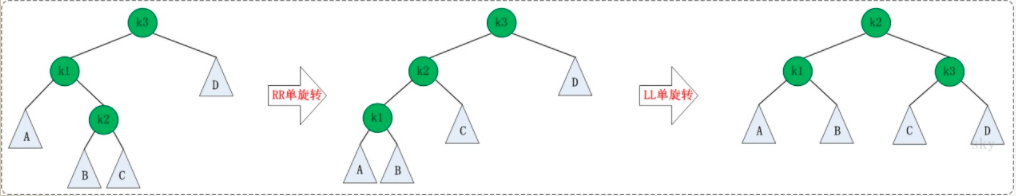
对于RR:可以通过一次旋转让整个树恢复平衡,旋转方法与LL对称。步骤：

1. 将原根节点的右子节点作为新的根节点；
2. 将新根节点的左子节点作为原根节点的右子节点；
3. 将原根节点作为新根节点的左子节点。



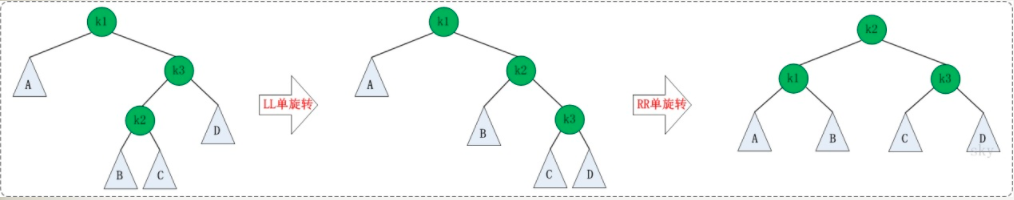
对于LR:需要进行两次旋转才能达到平衡，步骤：

1. 对根节点的左子节点进行RR旋转；
2. 对根节点进行LL旋转。



对于RL:需要进行两次旋转才能达到平衡，步骤：

1. 对根节点的右子节点进行LL旋转；
2. 对根节点进行RR旋转。



1. 平衡多路查找树(B-Tree)：

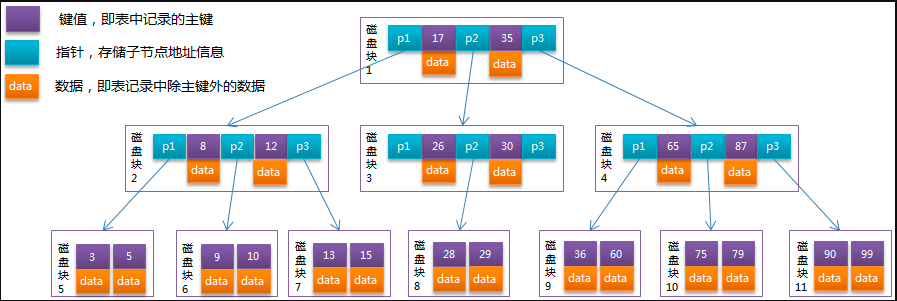
系统从磁盘读取数据到内存是以磁盘块为最小单位读取的，即位于同一个磁盘块的数据会一次性被读取到，而不是需要什么读取什么。

InnoDB存储引擎中有页(Page)的概念，页是其磁盘管理的最小单位。InnoDB

存储引擎中每个页的默认大小是16KB,可以修改参数innodb\_page\_size设置页的大小，mysql中可以通过命令show variables like 'innodb\_page\_size';查看页的大小。

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读取到内存是会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。所以B-Tree结构就是为了实现这种效果,B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。

https://www.cnblogs.com/vianzhang/p/7922426.html



**数据库：**

1. 存储引擎：

主流的有MyIsam、Innodb、Memery、Archive等

Innodb:

底层结构为B+树，

**守护线程：**

为用户线程提供公共服务，在没有用户线程可服务时会自动离开，守护线程的优先级比较低。用线程对象的setDaemon(true)将线程设置为守护线程。在守护线程中产生的新线程也是守护线程。

**锁：**

乐观锁：

Hadoop(分布式存储、分布式计算)

Common

Fdhs 分布式文件系统

存储并管理PB级数据



Yarn

MapReduce