2019
版

治愈系 Java 工程师面试指导课程

[Part1・JavaSE 基础]

[本教程涵盖了 JavaSE 部分大部分的面试题，建议大家将目录结构打开，对照目录结构做一个复习提纲准备面试。]

1 Java 基础知识

1.1 JDK/JRE/JVM 三者之间的联系与区别

JDK: 开发者提供的开发工具箱,是给程序开发者用的。它包括完整的 JRE（Java Runtime Environment），Java 运行环境，还包含了其他供开发者使用的工具包。
JRE: Java Runtime Environment jvm 运行时所必须的包依赖的环境都在 jre 中
JVM： 当我们运行一个程序时，JVM 负责将字节码转换为特定机器代码，JVM 提供了内存管理/垃圾回收和安全机制等。这种独立于硬件和操作系统，正是 java 程序可以一次编写多处执行的原因。
JDK > JRE > JVM

1.2 Java 面向对象编程三大特性

1.2.1 封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

1.2.2 继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。
关于继承如下 3 点请记住：

- 子类拥有父类非 private 的属性和方法。
- 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
- 子类可以用自己的方式实现父类的方法。（以后介绍）。

1.2.3 多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。在 Java 中有两种形式可以实现多态：**继承**（多个子类对同一方法的重写）和**接口**（实现接口并覆盖接口中同一方法）。

1.3 面向对象和面向过程

1.3.1 面向过程

优点： 性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、

Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。

缺点： 没有面向对象易维护、易复用、易扩展

1.3.2 面向对象

优点： 易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点： 性能比面向过程低

1.4 Java 语言有哪些特点

- ✓ 简单易学；
- ✓ **面向对象**（封装，继承，多态）；
- ✓ **平台无关性**（ Java 虚拟机实现平台无关性）；
- ✓ 可靠性；
- ✓ 安全性；
- ✓ 支持多线程（ C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
- ✓ 支持网络编程并且很方便（ Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
- ✓ 编译与解释并存；

1.5 Java 和 C++的区别?

没学过 C++，不代表面试官也没学过

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
- Java 有自动内存管理机制，不需要程序员手动释放无用内存

1.6 Java 基本数据类型

	byte	short	int	long	double	float	char	boolean
字节大小	1	2	4	8	8	4	2	1
占位大小	8	16	32	64	64	32	16	8

1.7 成员变量与局部变量的区别

- 从语法形式上看:成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被

public,private,static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。

- 从变量在内存中的存储方式来看:如果成员变量是使用 `static` 修饰的，那么这个成员变量是属于类的，如果没有使用 `static` 修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。
- 从变量在内存中的生存时间上看:成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
- 成员变量如果没有被赋初值:则会自动以类型的默认值而赋值（一种情况例外被 final 修饰的成员变量也必须显示地赋值），而局部变量则不会自动赋值。

1.8 静态方法和实例方法有何不同

- 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
- 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制.

1.9 构造方法

1.9.1 特性

名字与类名相同；
没有返回值，但不能用 void 声明构造函数；
生成类的对象时自动执行，无需调用。

1.9.2 构造器 Constructor 是否可被 override

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以 Constructor 也就不能被 override（重写）,但是可以 overload（重载）,所以你可以看到一个类中有多个构造函数的情况。

1.9.3 父子关系的构造方法的执行顺序

1. 父类有无参构造器，子类才可以写无参构造器；父类有含参构造器，子类才可以写含参构造器
2. 构造器不能被继承、重写
3. 当进行无参构造时，先调用父类无参构造器，然后调用子类无参构造器；当进行含参构造时，先调用父类含参构造器，然后调用子类含参构造器。

1.10 this 和 super 关键字

- super 关键字用于从子类访问父类的变量和方法.也包含构造方法

- this 关键字用于引用类的当前实例。此关键字是可选的，这意味着如果上面的示例在不使用此关键字的情况下表现相同。 但是，使用此关键字可能会使代码更易读或易懂。this 也可以调用当前类的构造方法。
- super 调用父类中的其他构造方法时，调用时要放在构造方法的首行！this 调用本类中的其他构造方法时，也要放在首行。
- this、super 不能用在 static 方法中。因为被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享。而 this 代表对本类对象的引用，指向本类对象；而 super 代表对父类对象的引用，指向父类对象；所以， this 和 super 是属于对象范畴的东西，而静态方法是属于类范畴的东西。

1.11 重载和重写的区别

重载： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。
重写： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

1.12 String 和 StringBuffer、StringBuilder

1.12.1 可变性

简单的来说：String 类中使用 final 关键字字符数组保存字符串，private final char value[] ，所以 String 对象是不可变的。而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

1.12.2 线程安全性

String 中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

1.12.3 性能

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

1.12.4 对于三者使用的总结

1. 操作少量的数据 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

1.13 自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；
拆箱：将包装类型转换为基本数据类型；

1.14 hashCode 与 equals 和==

面试官可能会问你：“你重写过 hashcode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

1.14.1 hashCode()介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。另外需要注意的是： Object 的 hashcode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的 内存地址 转换为整数之后返回。
public native int hashCode();
散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

1.14.2 为什么要有 hashCode

我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：
当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

1.14.3 hashCode()与 equals()的相关规定

1. 如果两个对象相等，则 hashcode 一定也是相同的
2. 两个对象相等,对两个对象分别调用 equals 方法都返回 true
3. 两个对象有相同的 hashcode 值，它们也不一定是相等的
4. 因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖
5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

1.14.4 为什么两个对象有相同的 hashcode 值,它们也不一定是相等的?

因为 hashCode() 所使用的杂凑算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode）。我们刚刚也提到了 HashSet,如果 HashSet 在对比的时候，同样的 hashcode 有多个对象，它会使用 equals() 来判断是否真的相同。也就是说 hashcode 只是用来缩小查找成本。

1.14.1 == 与 equals

== ：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

情况 1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。

情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

说明：

String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

1.15 关于 final 关键字的一些总结

final 关键字主要用在三个地方：变量、方法、类。

1. 对于一个 final 变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
2. 当用 final 修饰一个类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。
3. 使用 final 方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的 Java 实现版本中，会将 final 方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的 Java 版本已经不需要使用 final 方法进行这些优化了）。类中所有的 private 方法都隐式地指定为 fianl。

1.16 接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现)，抽象类可以有非抽象的方法
2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的

抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

备注:在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，必须重写，不然会报错。

1.17 static 关键字

static 关键字主要有以下四种使用场景：

1.17.1 修饰成员变量和成员方法

被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被 static 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。

调用格式：

类名.静态变量名

类名.静态方法名()

1.17.2 静态代码块

静态代码块定义在类中方法外，静态代码块在非静态代码块之前执行(静态代码块—非静态代码块—构造方法)。 该类不管创建多少对象，静态代码块只执行一次.

静态代码块的格式是:static { 语句体; }

一个类中的静态代码块可以有多个，位置可以随便放，它不在任何的方法体内，JVM 加载类时会执行这些静态的代码块，如果静态代码块有多个，JVM 将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。

静态代码块对于定义在它之后的静态变量，可以赋值，但是不能访问.

静态代码块定义在类中方法外，静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。该类不管创建多少对象，静态代码块只执行一次.

1.17.3 静态内部类（static 修饰类的话只能修饰内部类）

静态内部类与非静态内部类之间存在一个最大的区别: 非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：

1. 它的创建是不需要依赖外围类的创建。
2. 它不能使用任何外围类的非 static 成员变量和方法。

1.17.4 静态导包(用来导入类中的静态资源，1.5 之后的新特性)

格式为：import static 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

1.17.5 静态方法和非静态方法

静态方法属于类本身，非静态方法属于从该类生成的每个对象。 如果您的方法执行的操作不依赖于其类的各个变量和方法，请将其设置为静态（这将使程序的占用空间更小）。 否则，它应该是非静态的。
总结：

- 在外部调用静态方法时，可以使用”类名.方法名”的方式，也可以使用”对象名.方法名”的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
- 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制

1.17.6 静态代码块和非静态代码块

static{}静态代码块与{}非静态代码块(构造代码块)
相同点： 都是在 JVM 加载类时且在构造方法执行之前执行，在类中都可以定义多个，定义多个时按定义的顺序执行，一般在代码块中对一些 static 变量进行赋值。
不同点： 静态代码块在非静态代码块之前执行(静态代码块—非静态代码块—构造方法)。静态代码块只在第一次 new 执行一次，之后不再执行，而非静态代码块在每 new 一次就执行一次。非静态代码块可在普通方法中定义(不过作用不大)；而静态代码块不行。

一般情况下,如果有些代码比如一些项目最常用的变量或对象必须在项目启动的时候就执行的时候,需要使用静态代码块,这种代码是主动执行的。如果我们想要设计不需要创建对象就可以调用类中的方法，例如：Arrays 类，Character 类，String 类等，就需要使用静态方法，两者的区别是 静态代码块是自动执行的而静态方法是被调用的时候才执行的.

1.17.7 非静态代码块和函数

非静态代码块与构造函数的区别是： 非静态代码块是给所有对象进行统一初始化，而构造函数是给对应的对象初始化，因为构造函数是可以多个的，运行哪个构造函数就会建立什么样的对象，但无论建立哪个对象，都会先执行相同的构造代码块。也就是说，构造代码块中定义的是不同对象共性的初始化内容。

1.17.8 在一个静态方法内调用一个非静态成员为什么是非法的

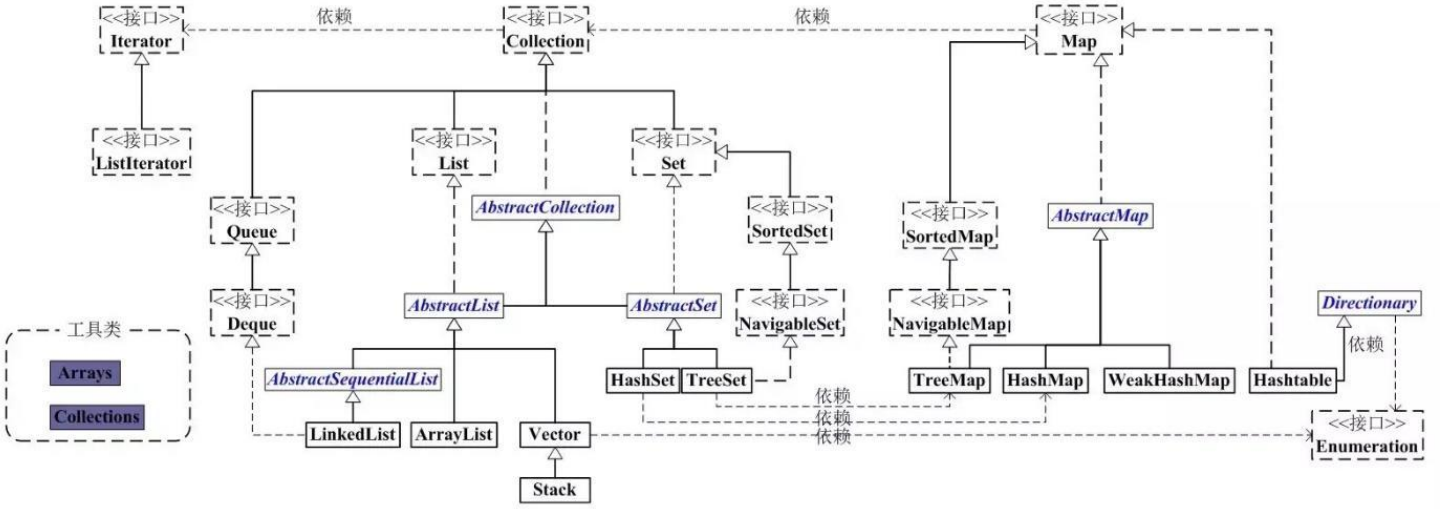
由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问非静态变量成员。

2 JAVA 常用集合

2.1 接口继承关系和实现

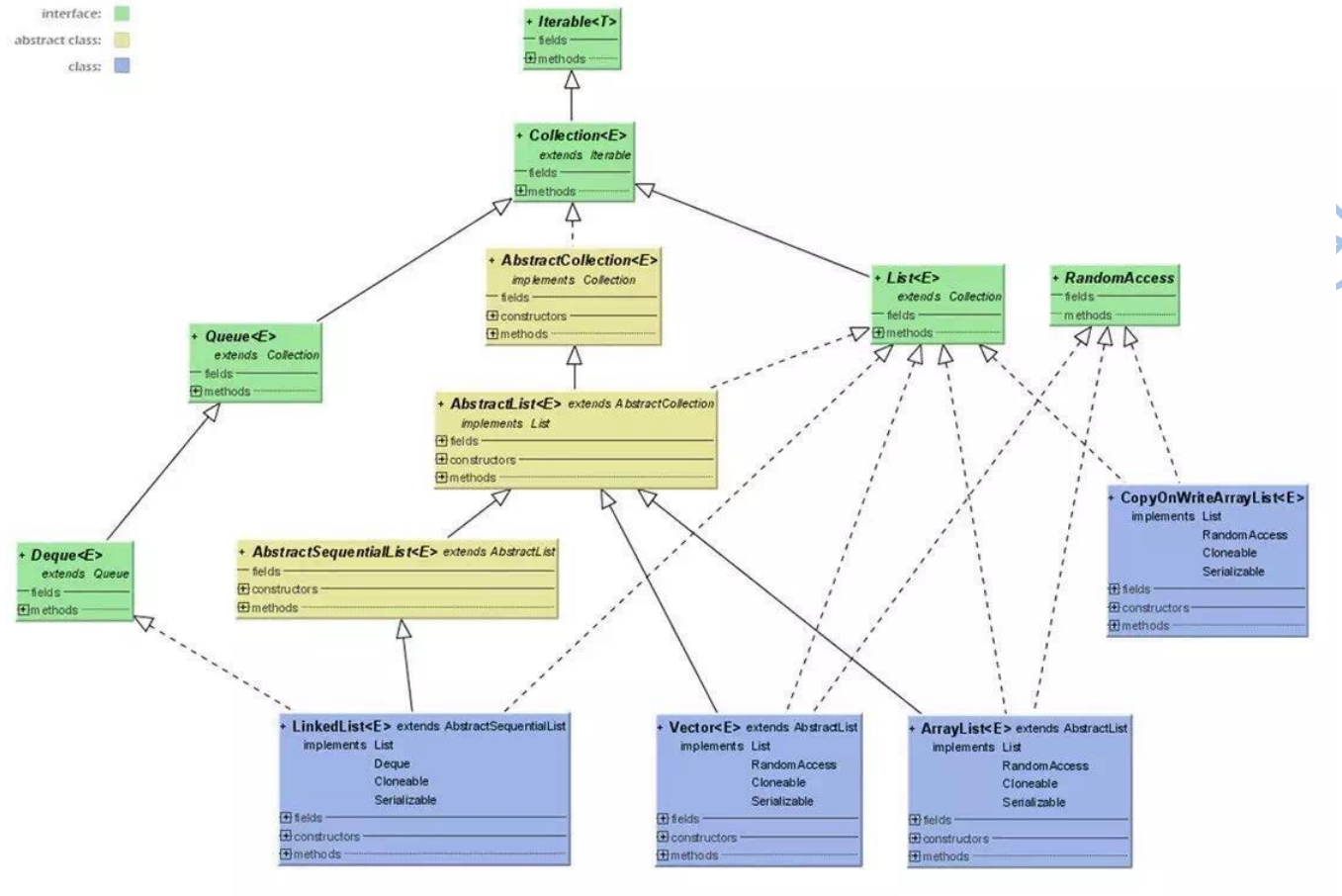
集合类存放于 Java.util 包中，主要有 3 种：set(集)、list(列表包含 Queue）和 map(映射)。

1. Collection：Collection 是集合 List、Set、Queue 的最基本的接口。
2. Iterator：迭代器，可以通过迭代器遍历集合中的数据
3. Map：是映射表的基础接口



2.2 List 接口

Java 的 List 是非常常用的数据类型。List 是有序的 Collection。Java List 一共三个实现类：分别是 ArrayList、Vector 和 LinkedList。



2.2.1 ArrayList（数组）

- ArrayList 是最常用的 List 实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要将已经有数 组的数据复制到新的存储空间中。当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进 行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。
- ArrayList 的底层是数组队列，相当于动态数组。与 Java 中的数组相比，它的容量能动态增长。在添加大量元素前，应用程序可以使用`ensureCapacity`操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。
- ArrayList 默认容量是 10，如果初始化时一开始指定了容量，或者通过集合作为元素，则容量为指定的大小或参数集合的大小。每次扩容为原来的 1.5 倍，如果新增后超过这个容量，则容量为新增后所需的最小容量。如果增加 0.5 倍后的新容量超过限制的容量，则用所需的最小容量与限制的容量进行判断，超过则指定为 Integer 的最大值，否则指定为限制容量大小。然后通过数组的复制将原数据复制到一个更大(新的容量大小)的数组。

2.2.2 Vector（数组实现、线程同步）

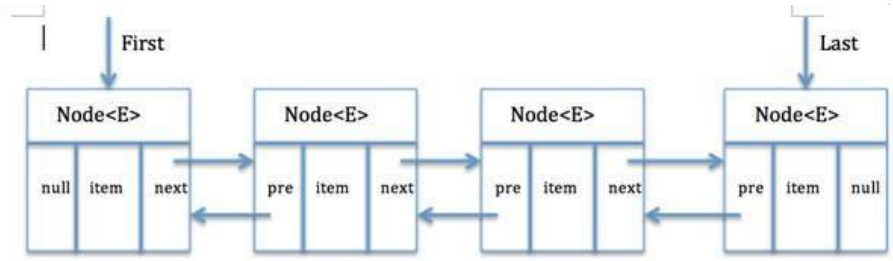
Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一 个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此， 访问它比访问 ArrayList 慢。

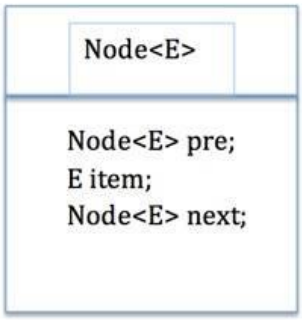
2.2.3 LinkList（链表）

LinkedList 是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较 慢。另外，他还提供了 List 接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆 栈、队列和双向队列使用。

LinkedList 是一个实现了 List 接口和 Deque 接口的双端链表。 LinkedList 底层的链表结构使它支持高效的插入和删除操作，另外它实现了 Deque 接口，使得 LinkedList 类也具有队列的特性; LinkedList 不是线程安全的，如果想使 LinkedList 变成线程安全的，可以调用静态类 Collections 类中的 synchronizedList 方法： java List list=Collections.synchronizedList(new LinkedList(...));

2.2.3.1 内部结构





2.2.4 Arraylist 与 LinkedList 异同

1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
2. 底层数据结构： Arraylist 底层使用的是 Object 数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别：）；
3. 插入和删除是否受元素位置的影响：

① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行 add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（ add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时

候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。

② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 get(int index) 方法)。

5. 内存空间占用： ArrayList 的空 间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

2.2.5 ArrayList 与 Vector 区别

Vector 类的所有方法都是同步的。可以由两个线程安全地访问一个 Vector 对象、但是一个线程访问 Vector 的话代码要在同步操作上耗费大量的时间。

Arraylist 不是同步的，所以在不需要保证线程安全时时建议使用 Arraylist。【CopyOnWriteArrayList 是同步的】。

2.2.6 System.arraycopy()和 Arrays.copyOf()

看两者源代码可以发现`copyOf()`内部调用了`System.arraycopy()`方法

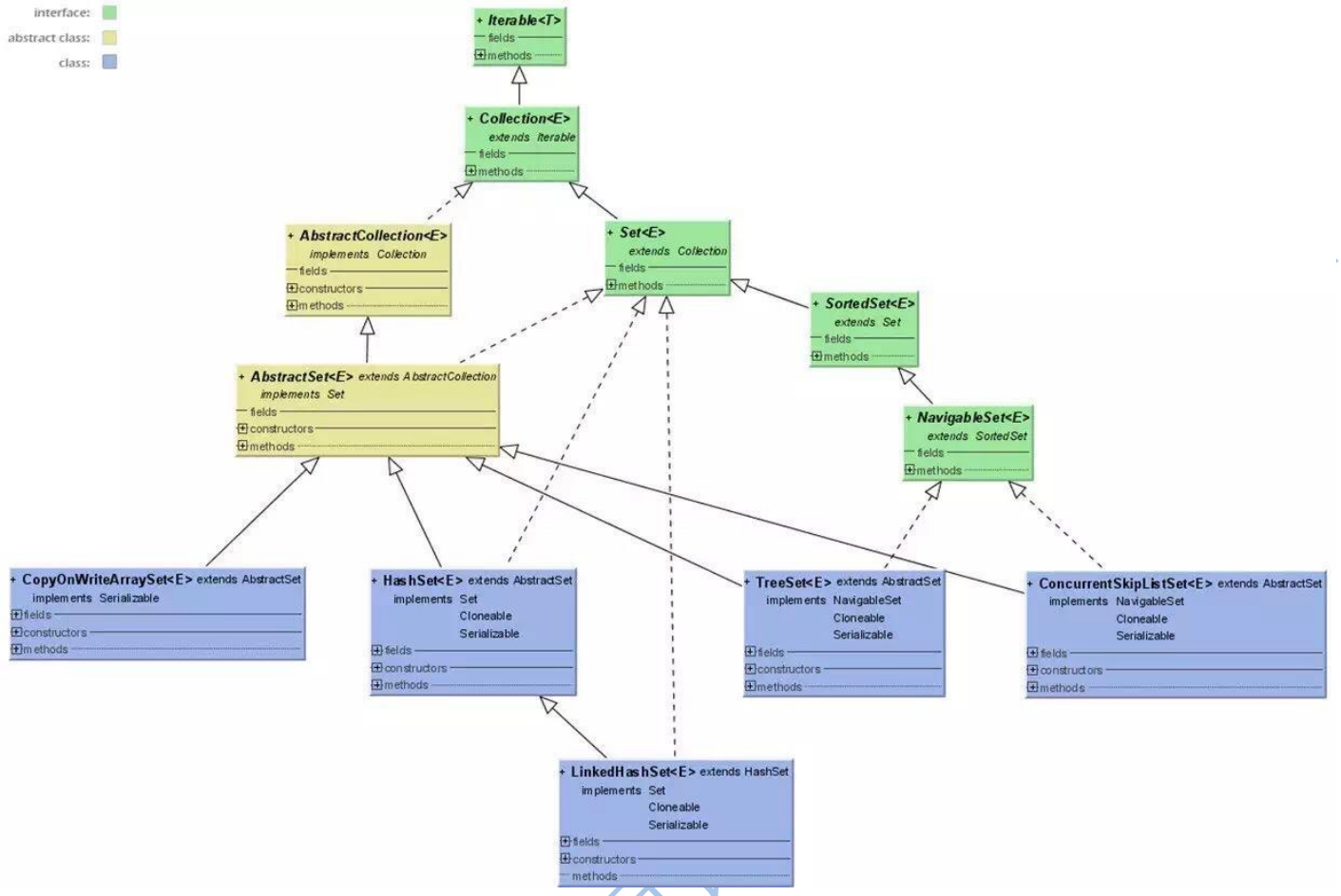
区别：

1. arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置

2. copyOf()是系统自动在内部新建一个数组，并返回该数组。

2.3 Set 接口

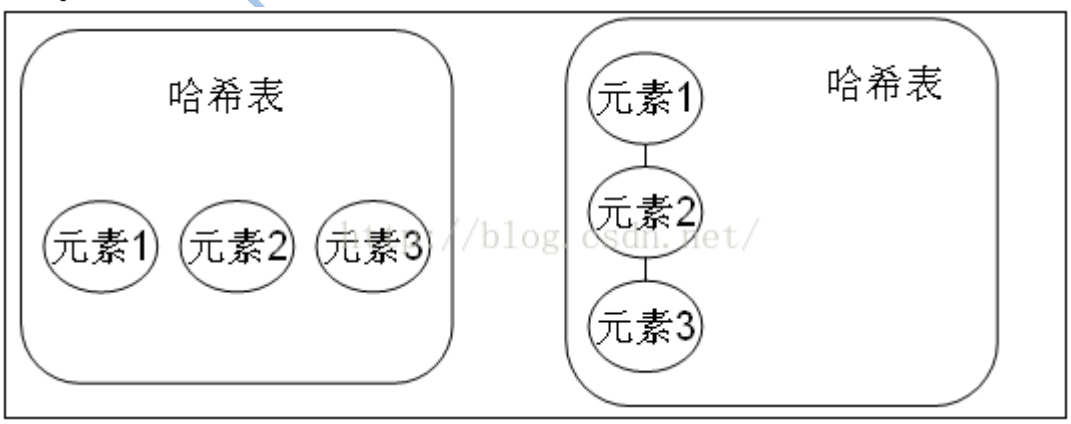
Set 注重独一无二的性质,该体系集合用于存储无序(存入和取出的顺序不一定相同)元素，值不能重 复。对象的相等性本质是对象 hashCode 值（java 是依据对象的内存地址计算出的此序号）判断 的，如果想要让两个不同的对象视为相等的，就必须覆盖 Object 的 hashCode 方法和 equals 方 法。



2.3.1 HashSet（Hash 表）

哈希表边存放的是哈希值。HashSet 存储元素的顺序并不是按照存入时的顺序（和 List 显然不 同） 而是按照哈希值来存的所以取数据也是按照哈希值取得。元素的哈希值是通过元素的 hashcode 方法来获取的, HashSet 首先判断两个元素的哈希值，如果哈希值一样，接着会比较 equals 方法 如果 equls 结果为 true ，HashSet 就视为同一个元素。如果 equals 为 false 就不是 同一个元素。

哈希值相同 equals 为 false 的元素是怎么存储呢,就是在同样的哈希值下顺延（可以认为哈希值相 同的元素放在一个哈希桶中）。也就是哈希一样的存一列。如图 1 表示 hashCode 值不相同的情 况；图 2 表示 hashCode 值相同，但 equals 不相同的情况。



HashSet 通过 hashCode 值来确定元素在内存中的位置。一个 hashCode 位置上可以存放多个元 素。

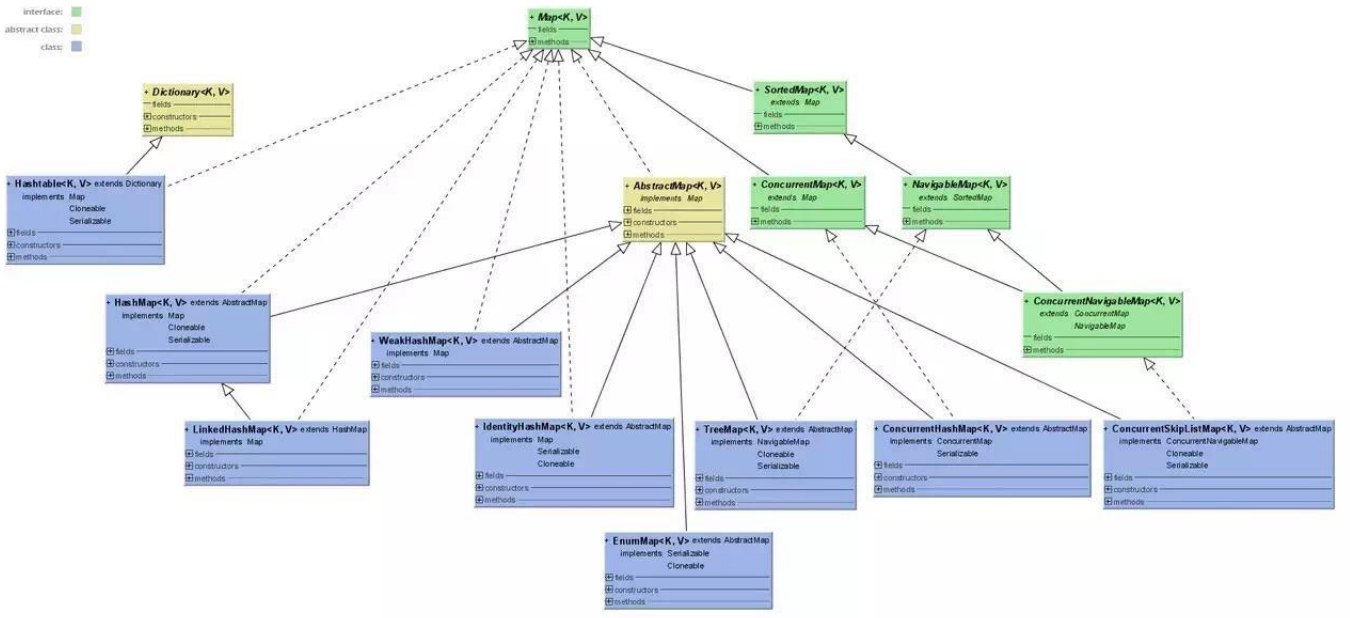
2.3.2 TreeSet（二叉树）

1. TreeSet()是使用二叉树的原理对新 add()的对象按照指定的顺序排序（升序、降序），每增 加一个对象都会进行排序，将对象插入的二叉树指定的位置。
2. Integer 和 String 对象都可以进行默认的 TreeSet 排序，而自定义类的对象是不可以的，自 己定义的类必须实现 Comparable 接口，并且覆写相应的 compareTo()函数，才可以正常使 用。
3. 在覆写 compare()函数时，要返回相应的值才能使 TreeSet 按照一定的规则来排序
4. 比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整 数、零或正整数。

2.3.3 LinkHashSet（HashSet+LinkedHashMap）

对于 LinkedHashSet 而 言 ， 它 继 承 与 HashSet 、 又 基 于 LinkedHashMap 来 实 现 的 。 LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法 操作上又与 HashSet 相同，因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并 通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操 作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。

2.4 Map



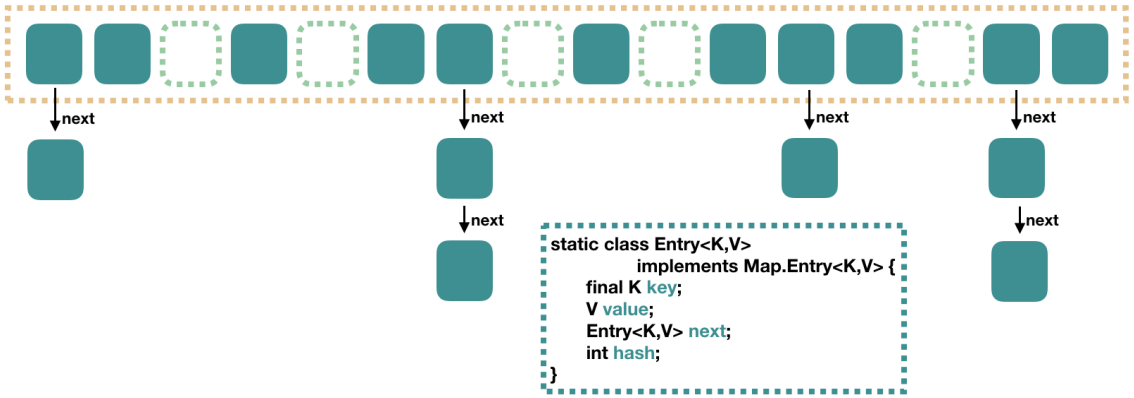
2.4.1 HashMap（数组+链表+红黑树）

HashMap 根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快 的访问速度，但遍历顺序却是不确定的。 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 非线程安全，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。如果需要满足线程安全，可以

用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。我们用下面这张图来介绍 HashMap 的结构。

2.4.1.1 Java7 的实现

Java7 HashMap 结构



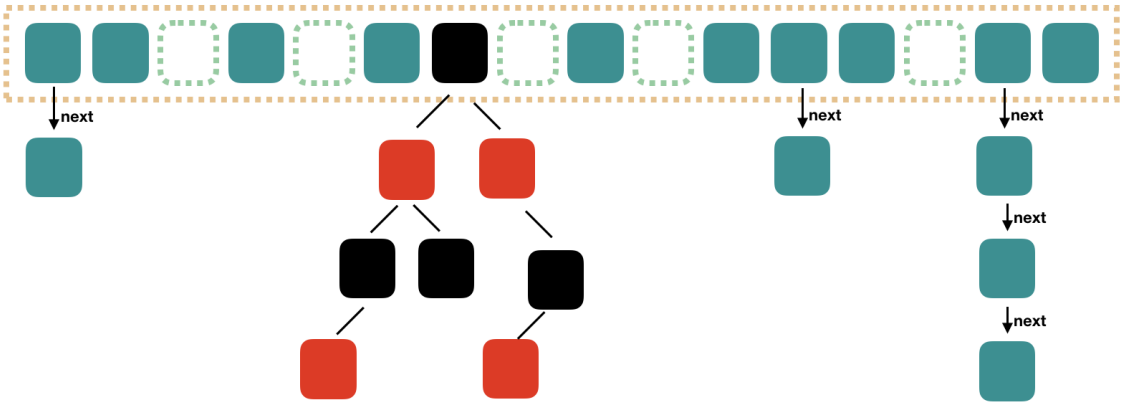
大方向上，HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。上图中，每个绿色 的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

1. capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。
2. loadFactor：负载因子，默认为 0.75。
3. threshold：扩容的阈值，等于 capacity * loadFactor

2.4.1.2 Java8 的实现

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑 树 组成。根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的 具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决 于链表的长度，为 O(n)。为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后， 会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

Java8 HashMap 结构



2.4.2 HashTable（线程安全）

Hashtable 是遗留类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类， 并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁。Hashtable 不建议在新代码中使用，不需要线程安全 的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换。

2.4.3 TreeMap（可排序）

TreeMap 实现 SortedMap 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。
如果使用排序的映射，建议使用 TreeMap。
在使 用 TreeMap 时，key 必须实 现 Comparable 接口或者在构 造 TreeMap 传入自定义 的 Comparator，否则会在运行时抛出 java.lang.ClassCastException 类型的异常。
参考：https://www.ibm.com/developerworks/cn/java/j-lo-tree/index.html

2.4.4 LinkHashMap（记录插入顺序）

LinkedHashMap 是 HashMap 的一个子类，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，按照访问次序排序。
参考 1：http://www.importnew.com/28263.html
参考 2：http://www.importnew.com/20386.html#comment-648123

2.4.5 HashMap 和 Hashtable 的区别

1. 线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. 效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. 对 Null key 和 Null value 的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。
4. 初始容量大小和每次扩充容量大小的不同 ：

①创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。

②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小,后面会介绍到为什么是 2 的幂次方。

5. 底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

2.4.6 HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n 代表数组长度）。这也就解释了 HashMap 的长度为什么是 2 的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：**“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是 2 的 n 次方；）。”并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。**

2.4.7 HashMap 多线程操作导致死循环问题

在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 resize()方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。

注意：jdk1.8 已经解决了死循环的问题。

2.4.8 HashSet 和 HashMap 区别

如果你看过 HashSet 源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了 clone() 方法、writeObject()方法、readObject()方法是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。）HashSet 就是 HashMap 的 key

HashMap	HashSet
实现了Map接口	实现Set接口
存储键值对	仅存储对象
调用put（）向map中添加元素	调用add（）方法向Set中添加元素
HashMap使用键（Key）计算Hashcode	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false
HashMap相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

2.5 ConcurrentHashMap 实现原理

由于 HashMap 是一个线程不安全的容器，主要体现在容量大于总量*负载因子发生扩容时会出现环形链表从而导致死循环。因此需要支持线程安全的并发容器 ConcurrentHashMap 。

2.5.1 Segment 段

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一 些。整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的 意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

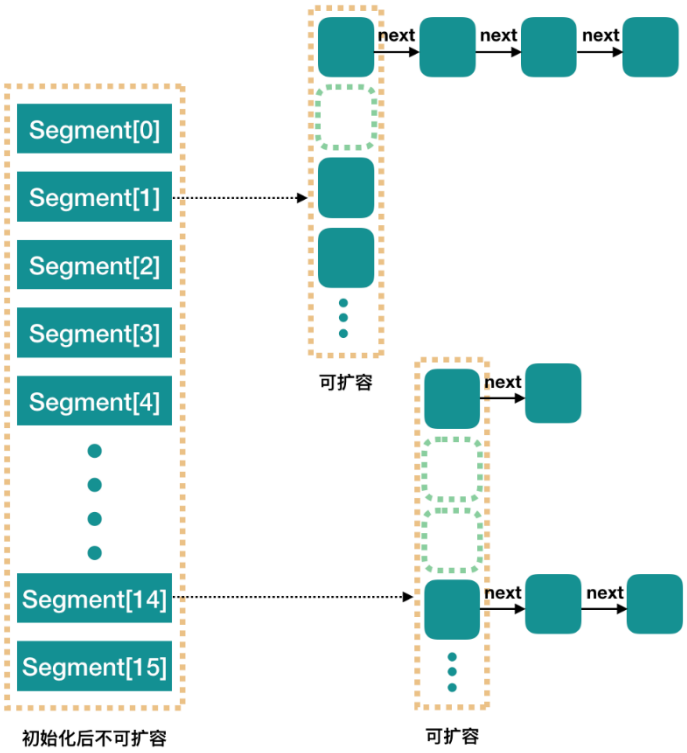
2.5.2 线程安全（Segment 继承 ReentrantLock 加锁）

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

2.5.3 JDK1.7 实现

2.5.3.1 Java7 的实现

Java7 ConcurrentHashMap 结构



concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

2.5.3.2 数据结构

由 Segment 数组、HashEntry 数组组成，和 HashMap 一样，仍然是数组加链表组成。

ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理，理论上 ConcurrentHashMap 支持 CurrencyLevel (Segment 数组数量)的线程并发。每当一个线程占用锁访问一个 Segment 时，不会影响到其他的 Segment。

2.5.3.3 get 方法

ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

只需要将 Key 通过 Hash 之后定位到具体的 Segment，再通过一次 Hash 定位到具体的元素上。由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。

2.5.3.4 put 方法

内部 HashEntry 类 ：

```
static final class HashEntry<K,V> {
    final int hash;
    final K key;
    volatile V value;
    volatile HashEntry<K,V> next;

    HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
```

虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。

首先也是通过 Key 的 Hash 定位到具体的 Segment，在 put 之前会进行一次扩容校验。这里比 HashMap 要好的一点是：HashMap 是插入元素之后再看是否需要扩容，有可能扩容之后后续就没有插入就浪费了本次扩容(扩容非常消耗性能)。而 ConcurrentHashMap 不一样，它是在将数据插入之前检查是否需要扩容，之后再做插入操作。

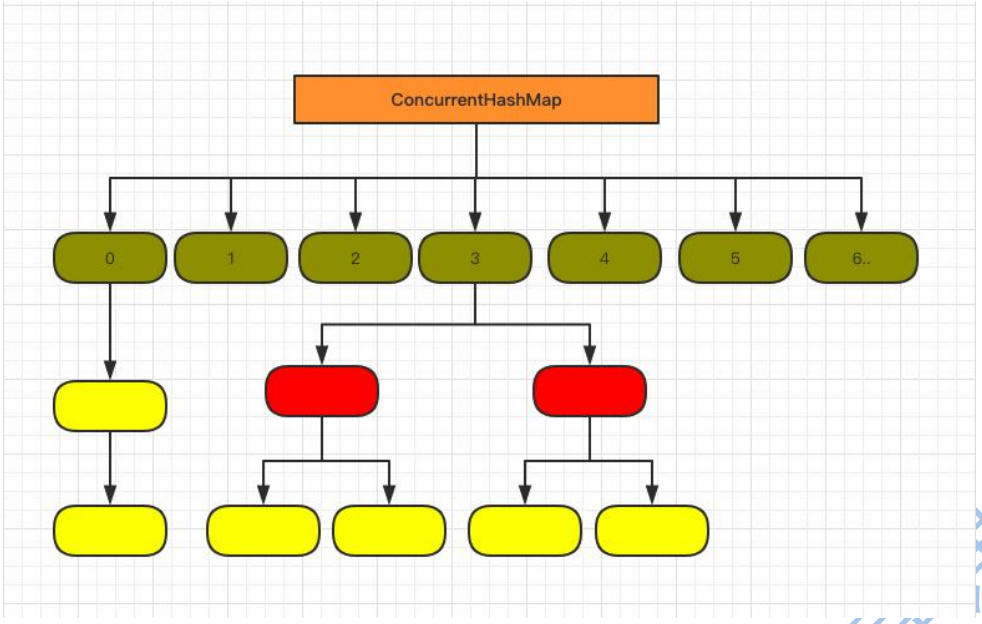
2.5.3.5 size 方法

每个 Segment 都有一个 volatile 修饰的全局变量 count ,求整个 ConcurrentHashMap 的 size 时很明显就是将所有的 count 累加即可。但是 volatile 修饰的变量却不能保证多线程的原子性，所有直接累加很容易出现并发问题。

但如果每次调用 size 方法将其余的修改操作加锁效率也很低。所以做法是先尝试两次将 count 累加，如果容器的 count 发生了变化再加锁来统计 size。

至于 ConcurrentHashMap 是如何知道在统计时大小发生了变化呢，每个 Segment 都有一个 modCount 变量，每当进行一次 put、remove 等操作，modCount 将会+1。只要 modCount 发生了变化就认为容器的大小也在发生变化。

2.5.4 JDK1.8 实现



1.8 中的 ConcurrentHashMap 数据结构和实现与 1.7 还是有着明显的差异。
其中抛弃了原有的 Segment 分段锁，而采用了 `CAS + synchronized` 来保证并发安全性。

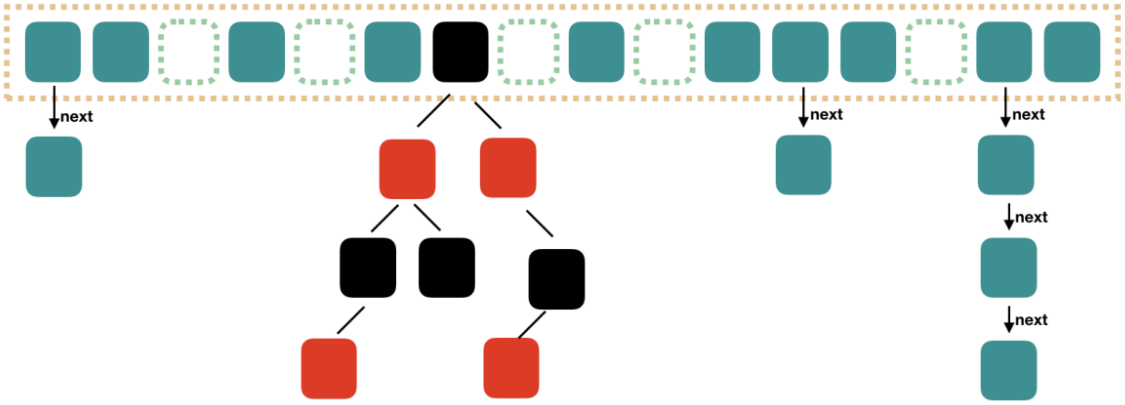
```
619  static class Node<K,V> implements Map.Entry<K,V> {
620      final int hash;
621      final K key;
622      volatile V val;
623      volatile Node<K,V> next;
624
625      Node(int hash, K key, V val, Node<K,V> next) {
626          this.hash = hash;
627          this.key = key;
628          this.val = val;
629          this.next = next;
630      }
631
632      public final K getKey()       { return key; }
633      public final V getValue()     { return val; }
634      public final int hashCode()   { return key.hashCode() ^ val.hashCode(); }
635      public final String toString(){ return key + "=" + val; }
636      public final V setValue(V value) { throw new UnsupportedOperationException(); }
639
```

也将 1.7 中存放数据的 HashEntry 改为 Node，但作用都是相同的。
其中的 `val next` 都用了 volatile 修饰，保证了可见性。

2.5.4.1 Java8 的实现

Java8 对 ConcurrentHashMap 进行了比较大的改动,Java8 也引入了红黑树。

Java8 ConcurrentHashMap 结构



2.5.4.2 put 方法

重点来看看 put 函数：

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) { ①
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0) ②
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) { ③
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED) ④
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) { ⑤
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD) ⑥
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
```

- 根据 key 计算出 hashcode 。
- 判断是否需要进行初始化。
- f 即为当前 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
- 如果当前位置的 hashcode==MOVED==-1,则需要进行扩容。
- 如果都不满足，则利用 synchronized 锁写入数据。
- 如果数量大于 TREEIFY_THRESHOLD 则要转换为红黑树。

2.5.4.3 get 方法

```
934 @ @ public V get(Object key) {
935         Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
936         int h = spread(key.hashCode());
937         if ((tab = table) != null && (n = tab.length) > 0 &&
938             (e = tabAt(tab, (n - 1) & h)) != null) {
939             if ((eh = e.hash) == h) {
940                 if ((ek = e.key) == key || (ek != null && key.equals(ek)))
941                     return e.val;
942             }
943             else if (eh < 0)
944                 return (p = e.find(h, key)) != null ? p.val : null;
945             while ((e = e.next) != null) {
946                 if (e.hash == h &&
947                     ((ek = e.key) == key || (ek != null && key.equals(ek))))
948                     return e.val;
949             }
950         }
951         return null;
952     }
```

- 根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值。
- 如果是红黑树那就按照树的方式获取值。
- 都不满足那就按照链表的方式遍历获取值。

2.5.5 总结

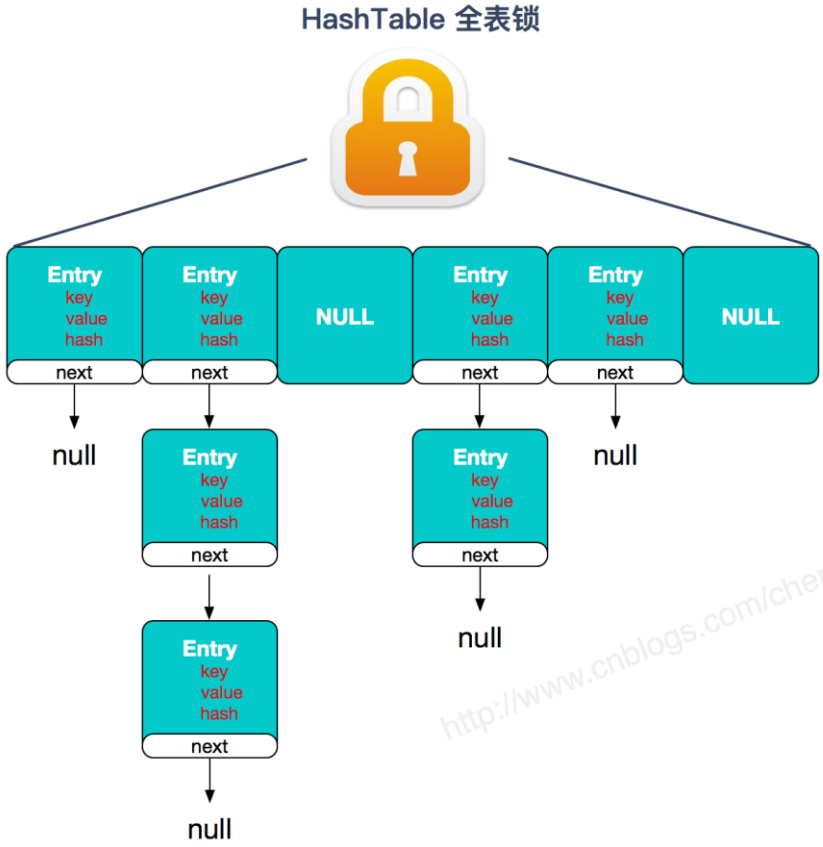
1.8 在 1.7 的数据结构上做了大的改动，采用红黑树之后可以保证查询效率（O(logn)），甚至取消了 ReentrantLock 改为了 synchronized，这样可以看出在新版的 JDK 中对 synchronized 优化是很到位的。
简单来说使用线程池有以下几个目的：

- 线程是稀缺资源，不能频繁的创建。
- 解耦作用；线程的创建于执行完全分开，方便维护。
- 应当将其放入一个池子中，可以给其他任务进行复用。

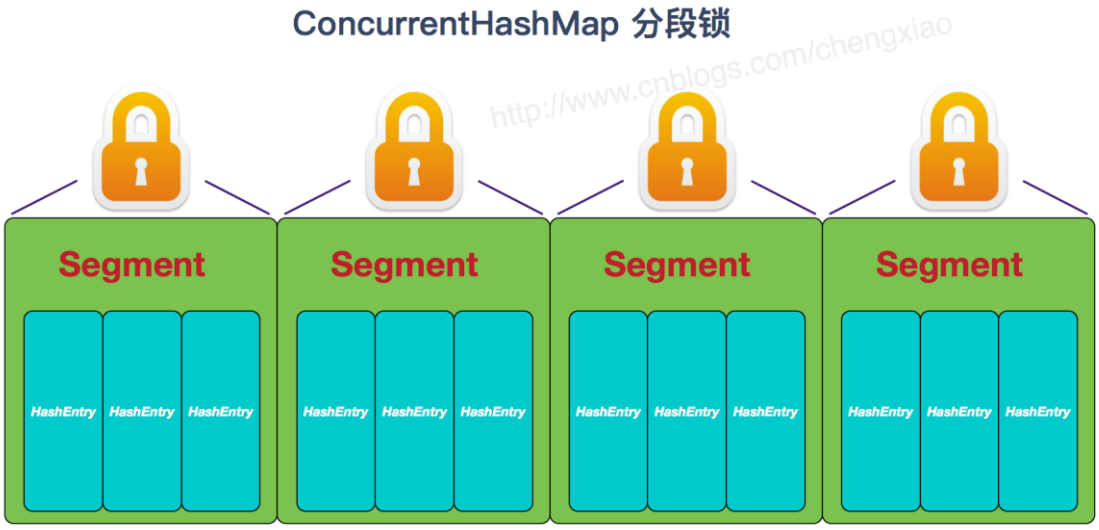
2.5.6 ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。
底层数据结构： JDK1.7 的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
实现线程安全的方式（重要）：
① 在 JDK1.7 的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6 以后 对 synchronized 锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；
② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。两者的对比图：

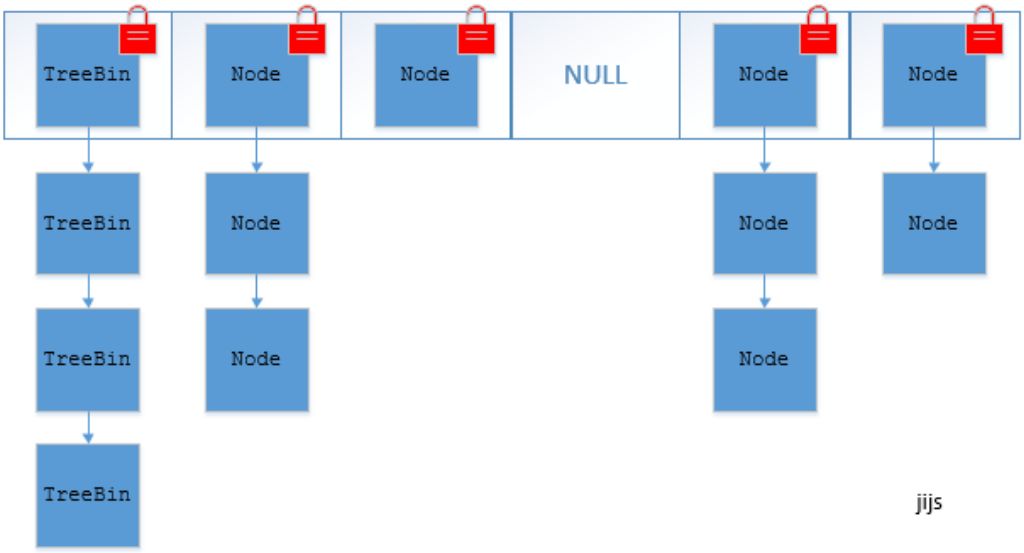
HashTable:



JDK1.7 的 ConcurrentHashMap：



JDK1.8 的 ConcurrentHashMap（TreeBin: 红黑二叉树节点 Node: 链表节点）：



2.6 Collections 工具类常用方法:

2.6.1 排序

void reverse(List list)//反转
void shuffle(List list)//随机排序
void sort(List list)//按自然排序的升序排序
void sort(List list, Comparator c)//定制排序，由 Comparator 控制排序逻辑
void swap(List list, int i , int j)//交换两个索引位置的元素
void rotate(List list, int distance)//旋转。当 distance 为正数时，将 list 后 distance 个元素整体移到前面。当 distance 为负数时，将 list 的前 distance 个元素整体移到后面。

2.6.2 查找,替换操作

int binarySearch(List list, Object key)//对 List 进行二分查找，返回索引，注意 List 必须是有序的
int max(Collection coll)//根据元素的自然顺序，返回最大的元素。
int min(Collection coll)
int max(Collection coll, Comparator c)//根据定制排序，返回最大元素，排序规则由 Comparatator 类控制。
int min(Collection coll, Comparator c) void fill(List list, Object obj)//用指定的元素代替指定 list 中所有元素。
int frequency(Collection c, Object o)//统计元素出现次数
int indexOfSubList(List list, List target)//统计 target 在 list 中第一次出现的索引，找不到则返回-1，
int lastIndexOfSubList(List source, list target).
boolean replaceAll(List list, Object oldVal, Object newVal), 用新元素替换旧元素

2.6.3 同步控制

Collections 提供了多个 synchronizedXxx()方法，该方法可以将指定集合包装成线程同步的集合，从而解决多线程并发访问集合时的线程安全问题。
我们知道 HashSet，TreeSet，ArrayList,LinkedList,HashMap,TreeMap 都是线程不安全的。Collections 提供了多个静态方法可以把他们包装成线程同步的集合。
synchronizedCollection(Collection<T> c) //返回指定 collection 支持的同步（线程安全的）collection。
synchronizedList(List<T> list) //返回指定列表支持的同步（线程安全的）List。
synchronizedMap(Map<K,V> m) //返回由指定映射支持的同步（线程安全的）Map。
synchronizedSet(Set<T> s) //返回指定 set 支持的同步（线程安全的）set。

2.6.4 Collections 设置不可变集合

emptyXxx()：返回一个空的、不可变的集合对象，此处的集合既可以是 List，也可以是 Set，还可以是 Map。
singletonXxx()：返回一个只包含指定对象（只有一个或一个元素）的不可变的集合对象，此处的集合可以是：List，Set，Map。
unmodifiableXxx()： 返回指定集合对象的不可变视图，此处的集合可以是：List，Set，Map。 上面三类方法的参数是原有的集合对象，返回值是该集合的”只读“版本。

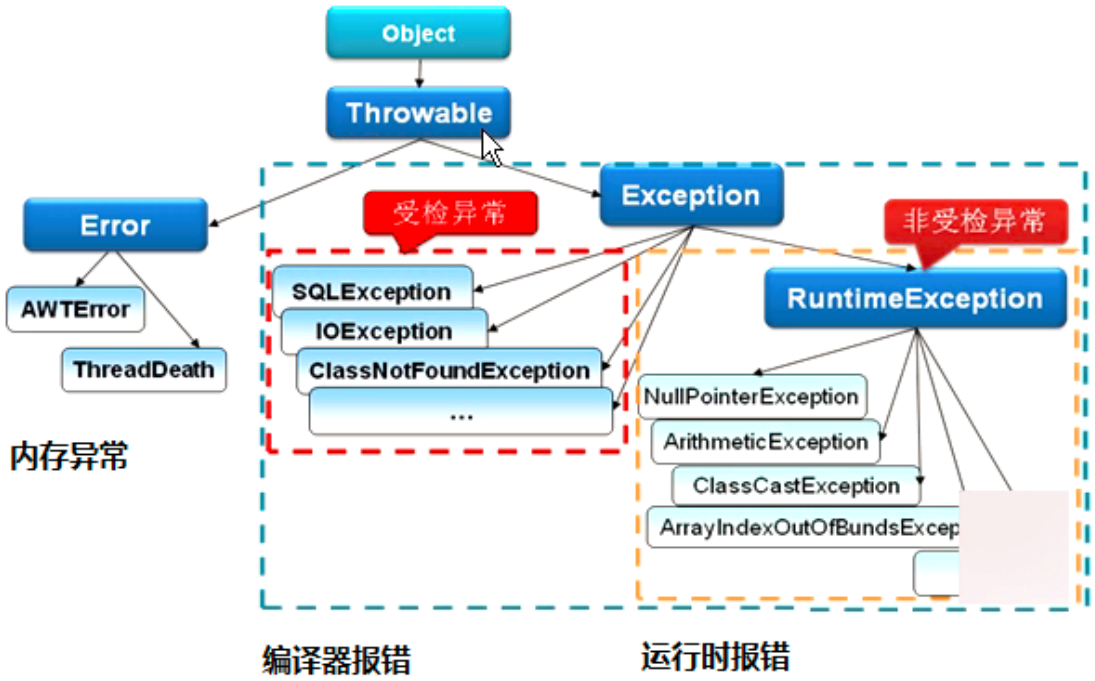
2.7 Arrays 类的常见操作

排序 : sort()
查找 : binarySearch()
比较: equals()
填充 : fill()
转列表: asList()
转字符串 : toString()
复制: copyOf()

3 JAVA 异常分类及处理

如果某个方法不能按照正常的途径完成任务，就可以通过另一种路径退出方法。在这种情况下 会抛出一个封装了错误信息的对象。此时，这个方法会立刻退出同时不返回任何值。另外，调用 这个方法的其他代码也无法继续执行，异常处理机制会将代码执行交给异常处理器。

3.1 异常的分类



Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

- Error 类是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果 出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

Exception（RuntimeException、CheckedException）

- 2. Exception 又有两个分支，一个是运行时异常 RuntimeException，一个是 CheckedException。

RuntimeException 如：NullPointerException、ClassCastException；一个是检查异常 CheckedException，如 I/O 错误导致的 IOException、SQLException。RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。如果出现 RuntimeException，那么一定是程序员的错误.

检查异常 CheckedException：一般是外部错误，这种异常都发生在编译阶段，Java 编译器会强 制程序去捕获此类异常，即会出现要求你把这段可能出现异常的程序进行 try catch，该类异常一般包括几个方面：

- ✓ 试图在文件尾部读取数据
- ✓ 试图打开一个错误格式的 URL
- ✓ 试图根据给定的字符串查找 class 对象，而这个字符串表示的类并不存在

3.2 异常的处理方式

3.2.1 抛出

遇到问题不进行具体处理，而是继续抛给调用者（throw,throws）

抛出异常有三种形式，一是 throw,一个 throws，还有一种系统自动抛异常

3.2.2 捕获

try catch 捕获异常针对性处理方式

3.2.3 Throw 和 throws 的区别

1. throws 用在函数上，后面跟的是异常类，可以跟多个；而 throw 用在函数内，后面跟的 是异常对象。
2. throws 用来声明异常，让调用者只知道该功能可能出现的问题，可以给出预先的处理方 式；throw 抛出具体的问题对象，执行到 throw，功能就已经结束了，跳转到调用者，并 将具体的问题对象抛给调用者。也就是说 throw 语句独立存在时，下面不要定义其他语 句，因为执行不到。
3. throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw 则是抛出了异常， 执行 throw 则一定抛出了某种异常对象。
4.两者都是消极处理异常的方式，只是抛出或者可能抛出异常，但是不会由函数去处理异 常，真正的处理异常由函数的上层调用处理。

4 Java 的流操作

4.1 BIO,NIO,AIO 有什么区别?

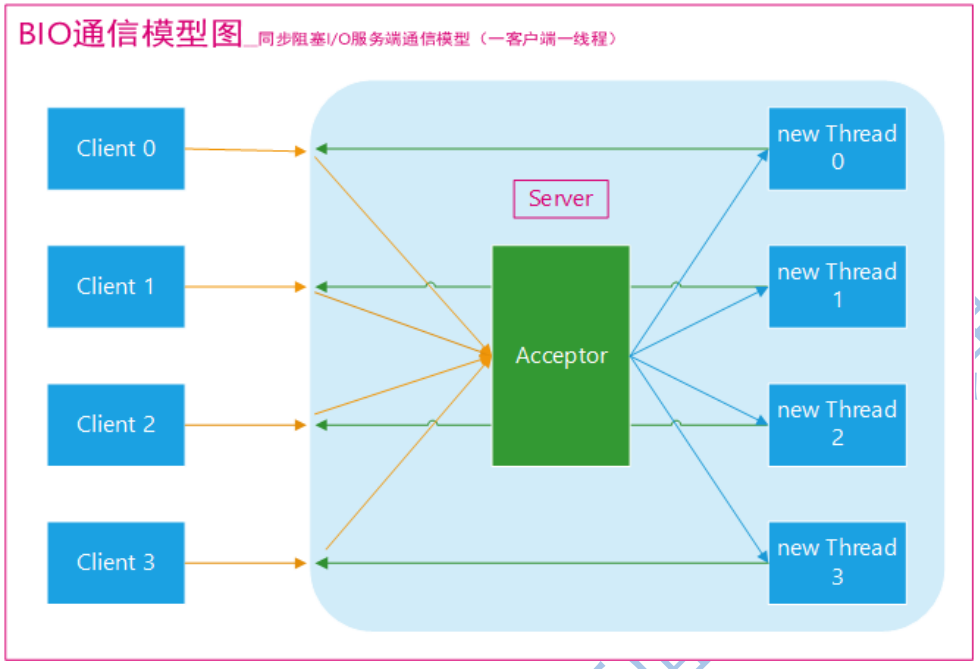
- **BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector，Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。 NIO 提供了与传统 BIO 模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

4.2 BIO (Blocking I/O)

同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。

4.2.1 传统 BIO

BIO 通信（一请求一应答）



采用 **BIO 通信模型** 的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接。我们一般通过在 `while(true)` 循环中服务端会调用 `accept()` 方法等待接收客户端的连接的方式监听请求，请求一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成， 不过可以通过多线程来支持多个客户端的连接，如上图所示。

如果要让 **BIO 通信模型** 能够同时处理多个客户端请求，就必须使用多线程（主要原因是 `socket.accept()`、`socket.read()`、`socket.write()` 涉及的三个主要函数都是同步阻塞的），也就是说它在接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的 **一请求一应答通信模型** 。我们可以设想一下如果这个连接不做任何事情的话就会造成不必要的线程开销，不过可以通过 **线程池机制** 改善，线程池还可以让线程的创建和回收成本相对较低。使用 `FixedThreadPool` 可以有效的控制了线程的最大数量，保证了系统有限的资源的控制，实现了 N(客户端请求数量):M(处理客户端请求的线程数量)的伪异步 I/O 模型（N 可以远远大于 M），下面一节"伪异步 BIO"中会详细介绍到。

我们再设想一下当客户端并发访问量增加后这种模型会出现什么问题？

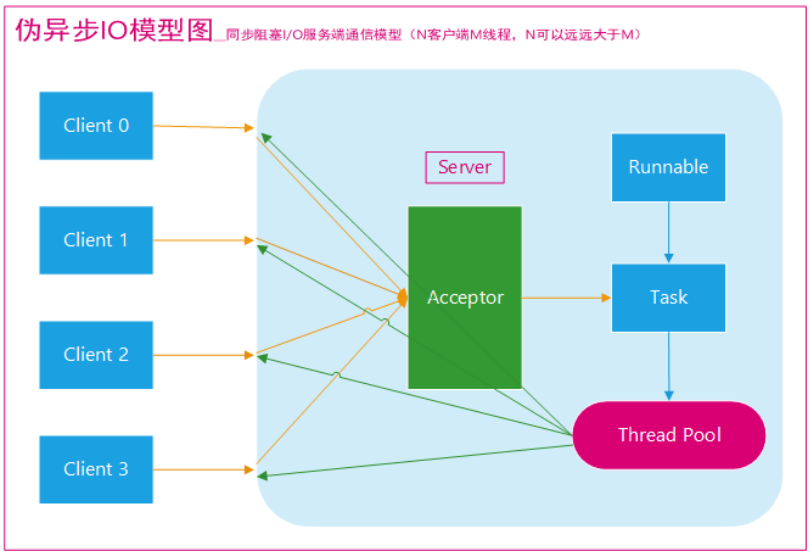
在 Java 虚拟机中，线程是宝贵的资源，线程的创建和销毁成本很高，除此之外，线程的切换成本也是很高的。尤其在 Linux 这样的操作系统中，线程本质上就是一个进程，创建和销毁线程都是重量级的系统函数。如果并发访问量增加会导致线程数急剧膨胀可能会导致线程堆栈溢出、创建新线程失败等问题，最终导致进程宕机或者僵死，不能对外提供服务。

4.2.2 伪异步 IO

为了解决同步阻塞 I/O 面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化——后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数 M：线程池最大线程数 N 的比例关系，其中 M 可

以远远大于 N.通过线程池可以灵活地调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

伪异步 IO 模型图



采用线程池和任务队列可以实现一种叫做伪异步的 I/O 通信框架，它的模型图如上图所示。当有新的客户端接入时，将客户端的 Socket 封装成一个 Task（该任务实现 java.lang.Runnable 接口）投递到后端的线程池中进行处理，JDK 的线程池维护一个消息队列和 N 个活跃线程，对消息队列中的任务进行处理。由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。

伪异步 I/O 通信框架采用了线程池实现，因此避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。不过因为它的底层仍然是同步阻塞的 BIO 模型，因此无法从根本上解决问题。

4.2.3 总结

在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

4.3 NIO 简介

Java NIO 是 **java 1.4** 之后新出的一套 IO 接口，这里的的新是相对于原有标准的 Java IO 和 Java Networking 接口。NIO 提供了一种完全不同的操作方式。

NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。

它支持面向缓冲的，基于通道的 I/O 操作方法。 随着 JDK 7 的推出，NIO 系统得到了扩展，为文件系统功能和文件处理提供了增强的支持。 由于 NIO 文件类支持的这些新的功能，NIO 被广泛应用于文件处理。

4.4 NIO 的特性/NIO 与 IO 区别[重点]

4.4.1 Channels and Buffers（通道和缓冲区）

IO 是面向流的，NIO 是面向缓冲区的

- 标准的 IO 编程接口是面向字节流和字符流的。而 NIO 是面向通道和缓冲区的，数据总是从通道中读到 buffer 缓冲区内，或者从 buffer 缓冲区写入到通道中；（ NIO 中的所有 I/O 操作都是通过一个通道开始的。）
- 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；
- Java NIO 是面向缓存的 I/O 方法。 将数据读入缓冲器，使用通道进一步处理数据。 在 NIO 中，使用通道和缓冲区来处理 I/O 操作。

4.4.2 Non-blocking IO（非阻塞 IO）

IO 流是阻塞的，NIO 流是不阻塞的。

- Java NIO 使我们可以进行非阻塞 IO 操作。比如说，单线程中从通道读取数据到 buffer，同时可以继续做别的事情，当数据读取到 buffer 中后，线程再继续处理数据。写数据也是一样的。另外，非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。
- Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了

4.4.3 Selectors（选择器）

NIO 有选择器，而 IO 没有。

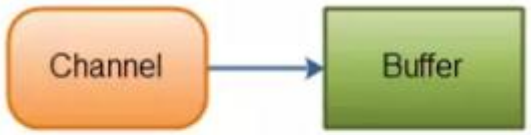
- 选择器用于使用单个线程处理多个通道。因此，它需要较少的线程来处理这些通道。
- 线程之间的切换对于操作系统来说是昂贵的。 因此，为了提高系统效率选择器是有用的。

4.4.4 读数据和写数据方式

通常来说 NIO 中的所有 IO 都是从 **Channel（通道）** 开始的。

从通道进行数据读取 ：创建一个缓冲区，然后请求通道读取数据。

从通道进行数据写入 ：创建一个缓冲区，填充数据，并要求通道写入数据。数据读取和写入操作图示：



4.5 NIO 核心组件简单介绍

NIO 包含下面几个核心的组件：

- Channels
- Buffers
- Selectors

整个 NIO 体系包含的类远远不止这三个，只能说这三个是 NIO 体系的“核心 API”。

4.5.1 通道

在 Java NIO 中，主要使用的通道如下（涵盖了 UDP 和 TCP 网络 IO，以及文件 IO）：

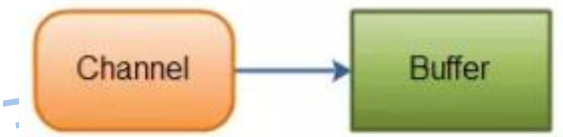
- DatagramChannel
- SocketChannel
- FileChannel
- ServerSocketChannel

4.5.1.1 Channel（通道）介绍

通常来说 NIO 中的所有 IO 都是从 Channel（通道） 开始的。

- **从通道进行数据读取** ：创建一个缓冲区，然后请求通道读取数据。
- **从通道进行数据写入** ：创建一个缓冲区，填充数据，并要求通道写入数据。

数据读取和写入操作图示：



4.5.1.2 Java NIO Channel 通道和流非常相似，主要有以下几点区别

通道可以读也可以写，流一般来说是单向的（只能读或者写，所以之前我们用流进行 IO 操作的时候需要分别创建一个输入流和一个输出流）。

通道可以异步读写。

通道总是基于缓冲区 Buffer 来读写。

4.5.1.3 Java NIO 中最重要的几个 Channel 的实现

- FileChannel： 用于文件的数据读写
- DatagramChannel： 用于 UDP 的数据读写
- SocketChannel： 用于 TCP 的数据读写，一般是客户端实现
- ServerSocketChannel: 允许我们监听 TCP 链接请求，每个请求会创建会一个 SocketChannel，一般是服务器实现

4.5.2 缓冲区

Java NIO Buffers 用于和 NIO Channel 交互。 我们从 Channel 中读取数据到 buffers 里，从 Buffer 把数据写入到 Channels.

Buffer 本质上就是一块内存区，可以用来写入数据，并在稍后读取出来。这块内存被 NIO Buffer 包裹起来，对外提供一系列的读写方便开发的接口。

在 Java NIO 中使用的核心缓冲区如下（覆盖了通过 I/O 发送的基本数据类型：byte, char、short, int, long, float, double ，long）：

- ByteBuffer
- CharBuffer
- ShortBuffer
- IntBuffer
- FloatBuffer
- DoubleBuffer
- LongBuffer

4.5.2.1 利用 Buffer 读写数据，通常遵循四个步骤

1. **把数据写入 buffer；**
2. **调用 flip；**
3. **从 Buffer 中读取数据；**
4. **调用 buffer.clear()或者 buffer.compact()。**

当写入数据到 buffer 中时，buffer 会记录已经写入的数据大小。当需要读数据时，通过 **flip()**方法把 buffer 从写模式调整为读模式；在读模式下，可以读取所有已经写入的数据。

当读取完数据后，需要清空 buffer，以满足后续写入操作。清空 buffer 有两种方式：调用 **clear()** 或 **compact()** 方法。**clear 会清空整个 buffer，compact 则只清空已读取的数据**，未被读取的数据会被移动到 buffer 的开始位置，写入位置则近跟着未读数据之后。

4.5.2.2 Buffer 的容量，位置，上限（Buffer Capacity, Position and Limit）

Buffer 缓冲区实质上就是一块内存，用于写入数据，也供后续再次读取数据。这块内存被 NIO Buffer 管理，并提供一系列的方法用于更简单的操作这块内存。

一个 Buffer 有三个属性是必须掌握的，分别是：

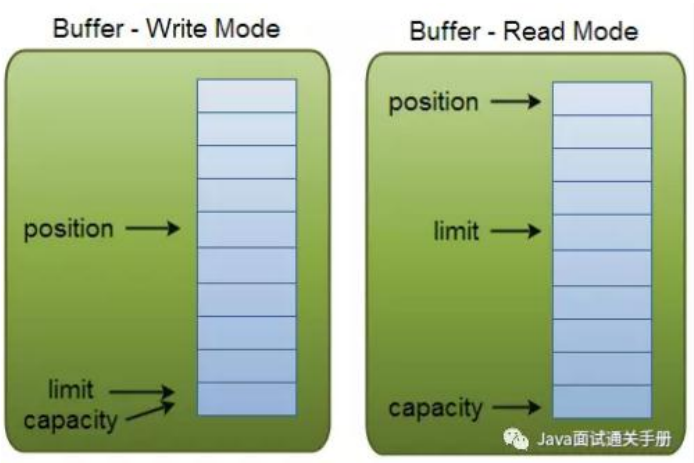
capacity 容量

position 位置

limit 限制

position 和 limit 的具体含义取决于当前 buffer 的模式。capacity 在两种模式下都表示容量。

读写模式下 position 和 limit 的含义：



4.5.2.3 容量（Capacity）

作为一块内存，buffer 有一个固定的大小，叫做 capacit（容量）。也就是最多只能写入容量值得字节，整形等数据。一旦 buffer 写满了就需要清空已读数据以便下次继续写入新的数据。

4.5.2.4 位置（Position）

当写入数据到 Buffer 的时候需要从一个确定的位置开始，默认初始化时这个位置 position 为 0，一旦写入了数据比如一个字节，整形数据，那么 position 的值就会指向数据之后的一个单元，position 最大可以到 capacity-1.

当从 Buffer 读取数据时，也需要从一个确定的位置开始。buffer 从写入模式变为读取模式时，position 会归零，每次读取后，position 向后移动。

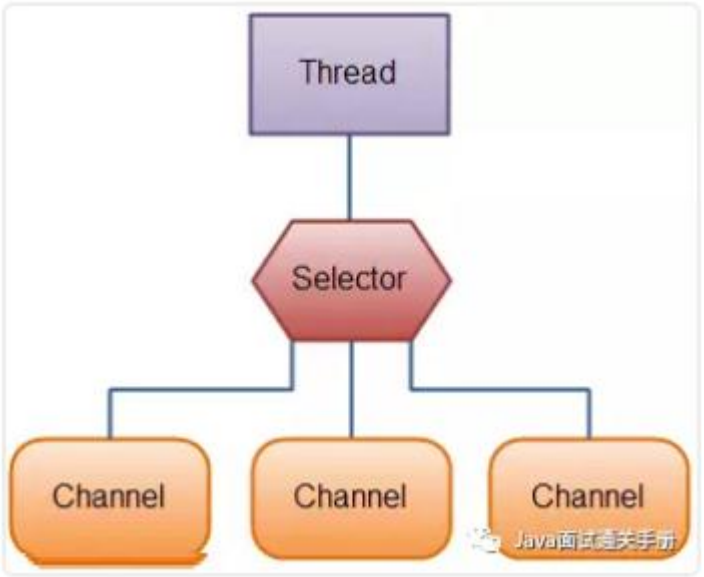
4.5.2.5 上限（Limit）

在写模式，limit 的含义是我们所能写入的最大数据量，它等同于 buffer 的容量。

一旦切换到读模式，limit 则代表我们所能读取的最大数据量，他的值等同于写模式下 position 的位置。换句话说，您可以读取与写入数量相同的字节数（限制设置为写入的字节数，由位置标记）。

4.5.3 选择器

Java NIO 提供了“选择器”的概念。这是一个可以用于监视多个通道的对象，如数据到达，连接打开等。因此，单线程可以监视多个通道中的数据。
如果应用程序有多个通道(连接)打开，但每个连接的流量都很低，则可考虑使用它。 例如：在聊天服务器中。
下面是一个单线程中 Slector 维护 3 个 Channel 的示意图：



要使用 Selector 的话，我们必须把 Channel 注册到 Selector 上，然后就可以调用 Selector 的 select()方法。这个方法会进入阻塞，直到有一个 channel 的状态符合条件。当方法返回后，线程可以处理这些事件。

Selector 一般称 为选择器 ，当然你也可以翻译为 **多路复用器** 。它是 Java NIO 核心组件中的一个，用于检查一个或多个 NIO Channel（通道）的状态是否处于可读、可写。如此可以实现单线程管理多个 channels,也就是可以管理多个网络链接。

使用 Selector 的好处在于： 使用更少的线程来就可以来处理通道了， 相比使用多个线程，避免了线程上下文切换带来的开销。

5 Java 反射

动态语言，是指程序在运行时可以改变其结构：新的函数可以引进，已有的函数可以被删除等结 构上的变化。比如常见的 JavaScript 就是动态语言，除此之外 Ruby,Python 等也属于动态语言， 而 C、C++则不属于动态语言。从反射角度说 JAVA 属于半动态语言

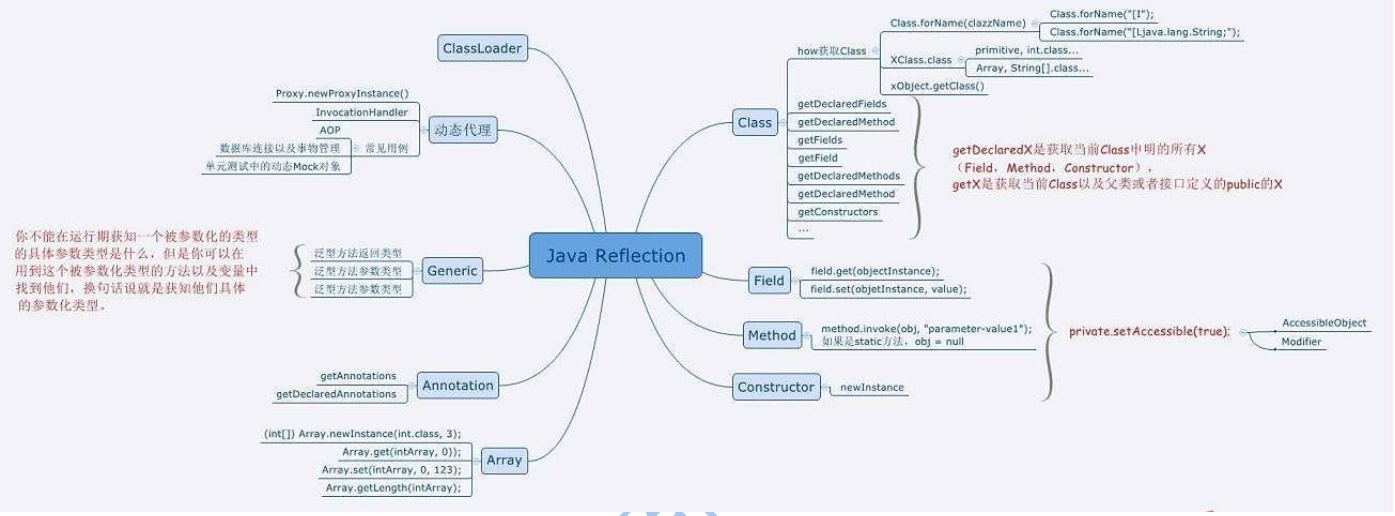
反射就是把 java 类中的各种成分映射成一个个的 Java 对象 例如：一个类有：成员变量、方法、构造方法、包等等信息，利用反射技术可以对一个类进行解剖，把个个组成部分映射成一个个对象。 （其实：一个类中这些成员方法、构造方法、在加入类中都有一个类来描述） 如图是类的正常加载过程：反射的原理在与 class 对象。 熟

悉一下加载的时候：Class 对象的由来是将 class 文件读入内存，并为之创建一个 Class 对象。 其中这个 Class 对象很特殊。我们先了解一下这个 Class 类

- JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；
- 对于任意一个对象，都能够调用它的任意一个方法和属性；
- 这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

5.1 反射机制概念

运行状态中知道类所有的属性和方法



在 Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法； 并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象方 法的功能成为 Java 语言的反射机制。

5.2 反射的应用场合

在 Java 程序中许多对象在运行是都会出现两种类型：编译时类型和运行时类型。编译时的类型由声明对象时实用的类型来决定，运行时的类型由实际赋值给对象的类型决定。如：
Person p=new Student()；其中编译时类型为 Person，运行时类型为 Student。

编译时类型无法获取具体方法程序在运行时还可能接收到外部传入的对象，该对象的编译时类型为 Object,但是程序有需要调用该对象的运行时类型的方法。为了解决这些问题，程序需要在运行时发现对象和类的真实信息。然而，如果编译时根本无法预知该对象和类属于哪些类，程序只能依靠运行时信息来发现该对象 和类的真实信息，此时就必须使用到反射了。

5.3 Java 反射 API

反射 API 用来生成 JVM 中的类、接口或则对象的信息。
1. Class 类：反射的核心类，可以获取类的属性，方法等信息。
2. Field 类：Java.lang.reflec 包中的类，表示类的成员变量，可以用来获取和设置类之中的属性值。
3. Method 类：Java.lang.reflec 包中的类，表示类的方法，它可以用来获取类中的方法信息或者执行方法。

4. Constructor 类：Java.lang.reflec 包中的类，表示类的构造方法。

5.3.1 反射使用步骤

1. 获取想要操作的类的 Class 对象，他是反射的核心，通过 Class 对象我们可以任意调用类的方法。
2. 调用 Class 类中的方法，既就是反射的使用阶段。
3. 使用反射 API 来操作这些信息。

5.3.2 获取 Class 对象的 3 种方法

- 调用某个对象的 getClass()方法
  Person p=new Person();
  Class clazz=p.getClass();
- 调用某个类的 class 属性来获取该类对应的 Class 对象
  Class clazz=Person.class;
- 使用 Class 类中的 forName()静态方法(最安全/性能最好)
  Class clazz=Class.forName("类的全路径"); (最常用)

当我们获得了想要操作的类的 Class 对象后，可以通过 Class 类中的方法获取并查看该类中的方法和属性。

5.3.3 创建对象的两种方法

Class 对象的 newInstance()
1.使用 Class 对象的 newInstance()方法来创建该 Class 对象对应类的实例，但是这种方法要求该 Class 对象对应的类有默认的空构造器。
调用 Constructor 对象的 newInstance()
2.先使用 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance()方法来创建 Class 对象对应类的实例,通过这种方法可以选定构造方法创建实例。

6 JAVA 内部类

Java 类中不仅可以定义变量和方法，还可以定义类，这样定义在类内部的类就被称为内部类。根 据定义的方式不同，内部类分为静态内部类，成员内部类，局部内部类，匿名内部类四种。

6.1 静态内部类

定义在类内部的静态类

```
public class Out {
    private static int a; private int b;
    public static class Inner {
        public void print() {
            System.out.println(a);
```

```
        }
    }
}
```

1、静态内部类可以访问外部类所有的静态变量和方法，即使是 private 的也一样。
2. 静态内部类和一般类一致，可以定义静态变量、方法，构造方法等。
3. 其它类使用静态内部类需要使用“外部类.静态内部类”方式，如下所示：Out.Inner inner = new Out.Inner();inner.print();
4. Java 集合类 HashMap 内部就有一个静态内部类 Entry。Entry 是 HashMap 存放元素的抽象， HashMap 内部维护 Entry 数组用了存放元素，但是 Entry 对使用者是透明的。像这种和外部 类关系密切的，且不依赖外部类实例的，都可以使用静态内部类。

6.2 成员内部类

定义在类内部的非静态类，就是成员内部类。成员内部类不能定义静态方法和变量（final 修饰的 除外）。这是因为成员内部类是非静态的，类初始化的时候先初始化静态成员，如果允许成员内 部类定义静态变量，那么成员内部类的静态变量初始化顺序是有歧义的。

```
public class Out {
    private static int a;
    private int b;

    public class Inner {
        public void print() {
            System.out.println(a);
            System.out.println(b);
        }
    }
}
```

6.3 局部内部类（定义在方法中的类）

定义在方法中的类，就是局部类。如果一个类只在某个方法中使用，则可以考虑使用局部类

```
public class Out {
    private static int a;
    private int b;
    public void test(final int c) { final int d = 1;
        class Inner {
            public void print() {
                System.out.println(c);
            }
        }
    }
}
```

6.4 匿名内部类

匿名内部类我们必须要继承一个父类或者实现一个接口，当然也仅能只继承一个父类或者实现一 个接口。同时它也是没有 class 关键字，这是因为匿名内部类是直接使用 new 来生成一个对象的引 用。

```
public abstract class Bird {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public abstract int fly();
}
```

```
public class Test {
    public void test(Bird bird){
        System.out.println(bird.getName() + "能够飞 " + bird.fly() + "米"); }
    public static void main(String[] args) {
        Test test = new Test();
        test.test(new Bird() {
            public int fly() {
                return 10000;
            }
            public String getName() {
                return "大雁";
            }
        });
    }
}
```

7 JAVA 序列化

7.1 序列化和反序列化

- Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时， 这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长。但在现实应用中， 就可能要求在 JVM 停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。 Java 对象序列化就能够帮助

我们实现该功能。

- 对象**序列化**是一个用于将对象状态转换为字节流的过程，可以将其保存到磁盘文件中或通过网络发送到任何其他程序；从字节流创建对象的相反的过程称为**反序列化**。java 中的序列化实现 Serializable 接口体现
- 使用 Java 对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的”状态”，即它的成员变量。由此可知，**对象序列化不会关注类中的静态变量**。
- 除了在持久化对象时会用到对象序列化之外，当使用 RMI(远程方法调用)，或在网络中传递对象时， 都会用到对象序列化。Java 序列化 API 为处理对象序列化提供了一个标准机制。

7.2 Serializable 实现序列化

- 在 Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化。
- 通过 ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化。writeObject 和 readObject 自定义序列化策略
- 在类中增加 writeObject 和 readObject 方法可以实现自定义序列化策略。
- 序列化并不保存静态变量
- 要想将父类对象也序列化，就需要让父类也实现 Serializable 接口。

7.3 序列化 ID

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个 类的序列化 ID 是否一致（就是 private static final long **serialVersionUID**）

7.4 transient 关键字

对于不想进行序列化的变量，使用 **transient** 关键字修饰。transient 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 transient 修饰的变量值不会被持久化和恢复。**transient 只能修饰变量，不能修饰类和方法**。

- 在变量声明前加上 transient 关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。
- 服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。