# Java基础思维脑

## JDK包含JRE包含JVM包含JIT

- 字节码在JVM中运行
- 将源代码编译成字节码并在JVM上运行的好处主要有以下几点
  - 平台无关性 — 字节码是平台无关的，Java程序可以在任何安装了JVM的系统上运行
  - 安全性 — JVM提供了安全管理器，可以在字节码执行时对其进行检查
  - 跨语言交互 — JVM不仅支持Java字节码的执行，还支持其他语言编写的字节码

## Java基本概念

### 基本数据类型

- 转换
  - 自动类型转换 — 当容量小的数据类型与容量大的数据类型的变量做运算时，结果会自动提升为容量大的数据类型。 — **byte,char,short-->int-->long-->float-->double**；**还有一个boolean**
  - 包装类
    - 将基本数据类型转化为对象，可以在需要对象的上下文中使用基本数据类型。
    - 8 种包装类都属于引用数据类型。看重记忆 Integer 和 Character
      - Integer 的 == 和 equals()
        - 使用 == 当你使用 == 来比较两个 Integer 实例时，你实际上是比较它们的引用是否指向同一对象。由于整数值在 -128 到 127 范围内时会缓存，使用 == 比较这个范围内的 Integer 对象可能会返回 true，但是对于超出此范围的值，== 比较将返回 false。即使这两个对象代表相同的数值
        - 使用 equals(): equals() 方法会比较两个 Integer 的数值，而不考虑对象引用。如果两个 Integer 实例代表相同的数值，equals() 方法将返回 true。
      - parseInt vs. valueOf — 这两个方法都可以从字符串创建 Integer 对象，但 parseInt 返回一个基本类型 int，而 valueOf 返回 Integer 对象。
    - 包装类提供了对象表示形式，允许 null 值，以及在泛型代码和集合中使用。
  - 强制类型转换 — 用到强转符，可能会导致精度缺失
- 自动装箱和拆箱
  - 自动装箱: 将基本类型自动转换为对应的包装类对象
  - 自动拆箱: 将包装类对象自动转换为对应的基本类型值。
  - 好处：简化了程序员在处理基本类型和包装类之间的转换工作
  - 性能问题 — 自动装箱和拆箱涉及创建对象和基本类型之间的转换，这可能影响性能，特别是在大量计算的场景中
  - 一些潜在问题 — 空指针异常 — 如果你尝试对一个 null 的 Integer 进行拆箱，它会导致 NullPointerException，因为 null 不能转换为一个原始的 int

### 引用数据类型

- 接口和抽象类
  - 共同点：二者用动的条件都成立时，最终结果才是 true
  - &和&&的区别 — 不同点：&&只要第一个条件不成立，就不会再去判断第二个条件，最终结果直接为 false，而&判断的是所有的条件。
  - equals() 与 == 的区别
    - ==可以判断基本类型,也可以判断引用类型 — 基本类型直接存储值
    - 引用类型则是指向存储所对象的引用（或内存地址） — 所有的类实例（包括数组和字符串）都是引用类型。
    - equals判断两个对象是否相等 — equals 重写：比的是值内容
    - equals 不重写：比的是地址
    - equals不能用在基本类型
  - hashcode （）和 equals 的区别 — Hashcode 用于返回对象的哈希码，用于快速查找对象
  - equals 用于判断两个对象引用是否相等
  - 重写equals为什么要重写hashcode — 因为 Java 中的对象默认的hashCode()方法是根据对象的内存地址生成散列码的，只重写equals的话，还是会破坏相等方法视为不同的对象。
- 关键字
  - This、final、finally、finalize
    - This 表示当前对象的引用
    - final 用于声明常量、方法或类不可被重写、修改或继承
    - finally 是异常处理之后，必须被执行的代码块
    - finalize 是对象在被垃圾回收之前调用
  - 访问修饰符（四种）
    - Private：同一类内可见。
    - Default：同一包内可见，不使用任何修饰符。
    - Protected：对同一包内的类和所有子类都可见。
    - Public：对所有的类都可见。
  - 抽象类不能被final修饰 — final修饰的类不能被继承,而抽象类本身必须要被继承的,相互冲突
  - static修饰的方法只能调用静态方法和变量 — 静态方法创建出来的时候,非静态资源还没有创建出来,所以静态方法只能调用静态方法和变量
  - 局部内部类和匿名内部类访问局部变量 — 因为生命周期不一致,局部变量保存在栈中,方法结束后,非final修饰的变量会被销毁,为什么变量必须要加上final — 销毁,而局部内部类对局部变量的引用仍然存在,调用局部变量的时候就会出错
- 数组
  - 数组（Array）：
    - 定义：数组是一种线性数据结构，用于存储一系列固定大小的元素，允许随机访问 — 提供快速的随机访问
    - 访问方式：可以通过索引直接访问任何位置的元素，时间复杂度为 O1
    - 大小固定：一旦创建，大小通常不可改变（静态数组），虽然有动态数组（如Java中的ArrayList），但内部机制仍基于数组
    - 用途：适合快速访问和处理固定数量的数据
  - 栈（Stack）：
    - 定义：栈是一种遵循后进先出（LIFO）原则的线性数据结构 — 提供对最新添加元素的快速访问
    - 访问方式：只能在栈顶添加或删除元素。添加元素称为 “入栈” （push），删除元素称为 “出栈” （pop）
    - 动态大小：栈可以在运行时根据需要扩展或缩减
    - 用途：适用于实现递归算法、回溯问题、历史记录、撤销操作等场景
- 枚举 — 枚举类型是一种特殊的类类型，它包含了一组固定的常量
- 记录（record）

## Java高级特性

### 泛型

- 允许在定义类和接口的时候使用类型参数（type parameter）。声明的类型参数在使用时用具体的类型来替换。
- 常用通配符
  - <?>：无界通配符，表示未知类型。
  - <? extends T>：上界通配符，表示类型参数是T类型或T的子类型。
  - <? super T>：下界通配符，表示类型参数是T类型或T的父类型
- 是如何实现的
  - 泛型擦除 — Java的泛型信息只在编译期有效，在运行时会进行类型擦除，这意味着Jvm是不会感知到泛型的，泛型参数会被擦除到它们的边界，比如List<String>和List<Integer>，在编译后会被擦除到类型为List。编译后泛型类型会被擦除为原始类型
  - 类型擦除的缺点
    - 1.泛型不可以重载
    - 2.泛型异常类不可以多次catch
    - 3.泛型类中的静态变量也只有一份，不会有多份

### 反射

- 什么是反射机制? — 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
- 反射机制优缺点
  - 反射机制允许程序在运行时访问、检测和修改它本身的类、接口或对象的属性
  - 缺点：性能差,比直接的java代码慢很多 — 因为JVM需要在运行时解析类的信息，这增加了额外的开销
  - 优点
    - 灵活性高 — 反射能让程序在运行时根据需要动态地操作和调用类及其成员，增加了代码的灵活性和适应性
    - 运行期类型的判断 — 利用反射，程序能在运行时确定对象的实际类型，从而实现更通用的处理逻辑
    - 动态加载类 — 反射允许程序在运行时根据需要加载类，无需提前在编译时知道所有类，提高了系统的可扩展性
- 反射机制的应用场景
  - 动态创建对象
  - 获取和设置类的成员变量值
  - 动态代理设计模式也采用了反射机制，
  - 还有我们日常使用的 Spring / SpringMVC / Mybatis 等框架也大 量使用到了反射机制
  - 例如模块化的开发，通过反射去调用对应的字节码；
- Java获取反射的三种方法
  - 1.通过new对象实现反射机制
  - 2.通过路径实现反射机制
  - 3.通过类名实现反射机制
- 反射机制执行的过程
  - 反射获取类实例
  - 反射获取方法
  - 调用method.invoke（）方法
- 反射的实现原理
  - 反射类及反射方法的获取，都是通过从列表中搜寻查找匹配的方法，所以查找性能会随类的大小方法多少而变化
  - 每个类都会有一个与之对应的Class实例，从而每个类都可以获取method反射方法，并作用到到其他实例身上
  - 反射也是考虑了线程安全的，放心使用
  - 反射使用软引用relectionData缓存class信息，避免每次重新从jvm获取带来的开销
  - 反射调用多次生成新代理Accessor, 而通过字节码生存的则考虑了卸载功能，所以会使用独立的类加载器
  - 当找到需要的方法，都会copy一份出来，而不是使用原来的实例，从而保证数据隔离；调度反射方法，最终是由jvm执行invoke0()执行
- 核心类
  - Class：代表类的实体，在运行时表示类和接口。
  - Method：代表类的方法。
  - Field：代表类的成员变量。
  - Constructor：代表类的构造器

### SPI机制

- Service Provider Interface（SPI）是一种服务发现机制。它允许Java程序动态地发现和加载实现特定服务或驱动，而无需对服务提供者进行硬编码
- 实现
  - 在META-INF/services目录下通过配置文件指定接口的实现类。
  - 使用ServiceLoader加载这些服务

### Java新特性概览 (JDK 1.8)

- Lambda表达式 — 提供了一种清晰简洁的方式来表示单方法接口（即功能接口）的实例
- Stream API — 新增了对数据集合处理的高效类库，支持序列和并行处理
- 新的日期时间API — 引入了全新的时间日期API，修复了旧版java.util.Date中的问题
  - 原本是可变的，线程不安全
  - 旧API的设计不一致，使用起来不直观。例如，月份从0开始计数（0代表一月）
  - 旧API缺乏对时区以及日期时间运算的支持
  - 新API添加了对时钟（Clock）的抽象，让测试时更容易地控制当前时间的模拟
- 接口的默认方法和静态方法 — 允许在接口中定义默认方法和静态方法
- Optional类 — 用于更好的处理null值情况，避免空指针异常

## Java集合

### 两个接口

#### 单列集合 - Collection

- List（允许重复）
  - 分类
    - ArrayList — 主要是用数组存储
      - 扩容机制 — 当添加元素超过ArrayList当前容量时，它会创建一个新的数组，并将旧数组的内容复制到这个新数组中，通常容量会增加到原来的1.5倍
      - 如何实现序列化？ — ArrayList可以被序列化，ArrayList 中存储数据的数组 elementData 是用 transient 修饰的，因为这个数组是动态扩展的，并不是所有的空间都被使用，因此就不需要所有的内容都被序列化
      - 如何实现线程安全 — 通过重写序列化和反序列化方法，使得可以只序列化数组中有内容的那部分数据
        - CopyOnWriteArrayList — 通过写时复制（copy-on-write）的方式来避免并发冲突 — 在每次修改时，都会先复制出一个新的数组，然后在新的数组上修改，最后将内部的数组引用指向新数组
        - Collections.synchronizedList — 适用于读多写少的并发场景
        - 该方法可以创建一个线程安全的 List 包装器。它返回一个包装了原始 ArrayList 的 List 对象，所有访问都通过一个内置的锁进行同步
    - LinkedList — 主要是用链表存储 — 链表的优缺点
      - 优点 — 插入删除速度快 — 因为有next指针指向其下一个节点，通过改变指针的指向可以方便的增加 删除元素
      - 内存利用率高，不会浪费内存 — 可以使用内存中细小的不连续空间（大于node节点的大小），并且在需要空间的时候才创建空间）
      - 大小没有固定，拓展很灵活。
      - 缺点 — 不能随机查找，必须从第一个开始遍历，查找效率低
    - Vector — 就比Arraylist多了个 synchronized（线程安全），因为单线程环境下效率较低，现在已经不太建议使用
  - ArrayList和LinkedList区别
    - 是否保证线程安全 — 都不保证线程安全
    - 底层数据结构 — Arraylist 底层使用的是数组 — LinkedList 底层使用的是双链表
    - 插入和删除是否受元素位置的影响 — 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 — LinkedList 采用链表存储，所以插入、删除元素的时间复杂度不受元素位置的影响
    - 是否支持快速随机访问 — LinkedList 不支持高效的随机元素访问 — ArrayList 实现了RandmoAccess 接口，所以有随机访问功能
    - 内存空间占用 — ArrayList的空间浪费主要体现在会预留一定的容量空间 — 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
  - 如何实现数组和 List 之间的转换
    - 数组转 List：使用 Arrays. asList(array) 进行转换
    - List 转数组：使用 List 自带的 toArray() 方法
    - 数组转set：使用Hutool函数，直接传入数组new HashSet(arrays[])
  - 遍历一个 List 有哪些不同的方式
    - for 循环遍历，基于计数器 — 在集合外部维护一个计数器，然后依次读取每一个位置的元素， 当读取到最后一个元素后停止
    - 迭代器遍历，Iterator，Iterator 是面向对象的一个设计模式，目的是屏蔽不同数据集合的特点，统一遍历集合的接口。Java 在 Collections 中支持了 Iterator 模式
    - foreach 循环遍历，foreach 内部也是采用了 Iterator 的方式实现，使用时不需要显式声明 Iterator 或计数器。 — 优点是代码简洁，不易出错 — 缺点是只能做简单的遍历，不能在遍历过 程中操作数据集合，例如删除、替换
- Set不包含重复
  - HashSet — LinkedHashSet — HashSet 是基于 HashMap 实现的，HashSet的值存放于HashMap的key上，HashMap的value统一为present，因此 HashSet 的实现比较简单
  - TreeSet — TreeSet怎么对集合中的元素进行排序 — TreeSet底层数据结构是二叉树
    - 让元素自身具备比较性，需要元素对象实现Comparable接口，覆盖compareTo方法
    - 让集合自身具备比较性，需要定义一个实现了Comparator接口的比较器，覆盖compare方法
- Queue和Deque（Deque是Queue的特殊实现） — Queue 接口的实现，如 PriorityQueue、LinkedList（也可以作为队列使用） — Deque 接口的实现，如 ArrayDeque、LinkedList（也可以作为双端队列使用）
- Collection和Collections区别 — Collection是集合类的上级接口，继承它的接口主要有set和list — Collections是工具类，有很多操作集合的方法

#### 如何实现对集合数据排序

- 使用Stream中的sorted方法
- 使用Collections中的sort方法
- 使用有序的集合，例如：TreeSet，TreeMap

#### 如何把一个线程不安全的集合转化为线程安全集合

- 向HashSet 中add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合equles 方法比较
- HashSet 中的add ()方法会使用HashMap 的put()方法。HashMap 的 key 是唯一的在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，所以不会重复

#### 双列集合 - Map

- 哈希表的Map实现类
  - Hashtable
    - 子类 — Properties
    - Hashtable不允许null键和null值，就比hashMap多了个synchronized (线程安全)，不建议使用
    - 如果线程安全的HashMap，去使用**ConcurrentHashMap** — 是Java5中支持高并发、高吞吐量的线程安全HashMap实现 — 详情往下看
  - HashMap
    - TreeMap — TreeMap不是基于哈希表的，而是基于红黑树的NavigableMap实现 — 它按照键的自然顺序进行排序，或者根据创建时提供的Comparator进行排序 — TreeMap提供了比HashMap更高的查询、删除和插入成本，但它允许按照顺序访问键值对
    - 子类 — LinkedHashMap — 它维护了一个双向链表来保持插入顺序或访问顺序 — 提供了与HashMap相似的性能，但同时还保持了插入或访问的顺序
    - 是否允许Null值? — 允许null key nullvalue，对于两个 null 值，HashMap 会视它们为相等 — 在 HashMap 的上下文中，键的相等性是通过 equals() 方法来判断的，而 null 只有一个，所以 null 作为键是相等的 — 但是，如果你在数据库的上下文中（如你提到的MySQL），null 值通常被视为未知，因此两个 null 值在比较时并不相等
    - 扩容机制 — 在添加元素的时候，当元素达到负载因子和数组长度的乘积时（默认负载因子是 0.75），触发扩容，会将旧的元素重新散列到新的更大的数组当中
    - 数据结构 — JDK1.8之前，数组+链表的结构 — 从JDK1.8开始，当链表长度超过一定阈值（默认为8）并且数组长度大于64时，链表会转化为红黑树，以此数据结构变为数组+链表+红黑树
      - 为什么选择红黑树 — 当HashMap中的某个桶（bucket）冲突过多时，其链表会变长，导致查找效率降低。接近O(N) — 红黑树在保持相对平衡的同时，插入和删除操作较为高效，最多只需进行几次旋转操作，从而在数据量大时提供更快速度
      - 为什么HashMap的初始长度是16 — 因为长度大小很容易导致map扩容影响性能。如果分配的大小的话又会浪费资源，所以就使用16作为初始大小
      - 减少hash碰撞 — 提高map查询效率 — 分配2小的正规繁矿容 — 分配之大浪费资源
    - 哈希算法 — HashMap通过计算键的哈希值来确定键值对在数组中的位置 — 具体的表引位置通常通过将哈希值与数组长度取模（或使用其他哈希分布算法）得到
    - 哈希冲突
      - 定义 — 在哈希表中，不同的关键字值对应到同一个存储位置的现象 — 简单来说，就是两个或多个不同的输入数据经过哈希函数处理后，得到了相同的哈希值，从而产生了冲突
      - 解决方法
        - 开放定址法 — 从冲突位置起，往后找没有冲突的位置，直到找到 — 如果一个数据项不能放在其“自然”位置上，会尝试表中的其他位置
        - 再哈希法 — 同时构造多个不同的哈希函数，当发生哈希冲突时，使用另一个哈希函数计算地址，直到不发生冲突为止
        - 链地址法 — 将所有哈希地址相同的记录都链接在同一链表中 — 这样，同一个哈希地址下的所有元素都保存在一个链表中，查找时需要遍历该链表
        - 建立公共溢出区 — 将哈希表分为基本表和溢出表，将发生冲突的元素都存放在溢出表中
    - HashMap 和TreeMap的区别 — 底层和速率 — Hash Map 是一个基于哈希表的无顺存储的键值对，复杂度是 O（1） — Tree map 是红黑树实现，是有序存储键值对，复杂度是 O（log n）
    - HashMap和Hashtable区别
      - hashMap去掉了HashTable 的contains方法，但是加上了containsValue（）和containsKey（）方法
      - hashTable同步的，而HashMap是非同步的，效率上比hashTable要高
      - hashMap允许空键值，而hashTable不允许
    - HashMap你经常用在那个地方
      - 电子商务的应用中使用HashMap作为缓存
      - 金融领域出于性能的考虑非常多的运用HashMap和ConcurrentHashMap
      - controller层向前台返回数据可以用map封装数据
      - mybatis中的map可以作为参数或者封装结果集
  - map集合的两种取出方式
    - 通过map.keyset，先得到map集合的键，再根据键得到value值
    - 通过map.entrySet直接得到键值对

#### 内存泄漏和内存溢出

- 内存泄露（memory leak） — 指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露似乎不会有大的影响，但内存泄露堆积后的后果就是内存的溢出。
- 内存溢出（out of memory）
  - 指程序申请内存时，没有足够的内存供申请者使用
  - 或者说，给了位一块存储 int 类型数据的存储空间，但是你却存储 long类型的数据，那么结果就是内存不够用，此时就会报错 OOM,即所谓的内存溢出

#### comparable 和 comparator的区别

- comparable接口实际上是出自java.lang包，它有一个 compareTo(Object obj)方法用来排序
- comparator接口实际上是出自 java.util 包，它有一个compare(Object obj1, Object obj2)方法用来排序

#### HashCode的方法和作用

- 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
- 在比较一个对象是否相等时，首先需要比较对象的hashcode是否相等，其次要比较equal是否相等

## Java面向对象

### 面向对象和面向过程

- 二者的区别 — 面向对象注重的是有哪些参与者，各自需要做什么。强调的是继承封装和多态 — 而面向过程注重的是事情的步骤与顺序，强调的是函数的顺序调用

### 面向对象的三大特征

- 继承 — 继承避免了对一般类和特殊类之间共同特征进行重复描述。即提高了代码的复用性
- 封装 — 封装说明一个类行为和属性与其他类的关系低耦合，高内聚，便代码便于修改，增强了代码的可维护性。同时封装好的类可以重复使用，也增强了代码的复用性
- 多态
  - 多态指的是类和类的关系，两个类有继承关系，有方法的重写，故而可以再调用有父类引用指向子类对象。多态必备三个要素：继承，重写，父类引用指向子类对象 — **重写和重载的区别**
    - 重写是子类对父类的方法 — 发生在父子类中 — 方法的名称参数必须相同 — 抛出的异常小于父类 — 访问修饰符大于等于父类
    - 重载是同一功能的不同实现方式 — 方法名称相同 参数类型不同 — 参数类型 参数数量 参数类型 参数顺序
    - 构造器不能被继承,所以不能被重写,但是可以被重载
  - 父类引用指向子类对象
- 父类非静态子类静态，谁先初始化（上到下）
  - 父类静态变量和静态初始化块。
  - 子类静态变量和静态初始化块。
  - 父类实例变量和初始化块。
  - 父类构造函数。
  - 子类实例变量和初始化块。
  - 子类构造函数。

### 类与对象

- Java 的对象的创建过程
  - 首先是类加载检查，然后是分配内存空间，然后是初始化默认值，然后设置对象头，然后执行构造方法
  - 创建对象的方式
    - 1 使用new关键字
    - 2 使用Class类的newInstance方法
    - 3 使用Constructor类的newInstance方法
    - 4 使用clone方法
      - clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法
      - 原因 — 使用 clone() 方法来拷贝一个对象即是负又有风险，它会抛出异常，并且还需要类型转换
      - clone() 的替代方案 — 可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象
    - 5 使用序列化和反序列化
- 深拷贝和浅拷贝
  - 浅拷贝创建一个新对象，但它包含的是对原始对象中包含的对象的引用，而不是对象本身的拷贝 — 一旦引用类型的数据被修改，所有共享这个数据的对象都会受到影响。这在多线程环境中可能会引起安全问题 — 由于浅拷贝的对象共享相同的引用数据，所以在不打算修改共享状态的情况下，也可能会不小心更改数据，这可能会导致难以跟踪的bug — 如果敏感数据通过引用类型共享，那么可能会因为对象之间的这种关联而被意外泄露出去
  - 深克隆则拷贝了所有，也就是说深克隆能够做到原对象和新对象之间完全没有影响。 — 增强的数据隔离 — 深拷贝不仅复制对象本身，还递归地复制对象中的所有引用类型属性，因此深拷贝对象是完全独立的 — 减少意外修改的风险 — 因此在不影响原始对象的情况下，可以放心地修改深拷贝对象的状态 — 保护敏感数据 — 使用深拷贝可以确保敏感数据不会通过引用被多个对象共享，从而降低数据泄露的风险
  - 使用序列化和反序列化也能完成**深复制**的功能 — 对象序列化后写入流中，此时也就不存在引用什么的概念，再从流中读取，生成新的对象，新对象和原对象之间也是完全互不影响的
  - 在Java中的实现：
    - 浅拷贝：对于Java中的简单数据类型和不可变对象，赋值操作本质上就是浅拷贝。如果需要对复杂对象进行浅拷贝，可以使用如Apache Commons Lang类中的SerializationUtils.clone()方法，或者手动创建一个新的对象并复制原始对象的字段值（不包括引用字段指向的对象）
    - 深拷贝：在Java中，深拷贝通常需要自定义实现。一种常见的方法是实现Cloneable接口并重写clone()方法，或者使用序列化和反序列化来创建对象的深拷贝。

### 接口与抽象类

- 区别
  - 接口是对于类行为的一种约束，是强调特定功能的一种实现
  - 抽象类是用于代码的复用，更多是定义抽象的概念
  - 类只能继承一个，而接口可以实现多个
  - 接口中的变量是静态变量，是不能被修改的，而抽象类中的变量默认是 default 修饰，可以在子类中重新定义或重新赋值
- 共同点
  - 都不能被实例化
  - 都包含抽象方法
  - 都有默认实现的一个方法
- 一个类如果有抽象方法，他必须是抽象类
- 接口内部的方法都默认是抽象的，除非它们被明确定义为 default 或 static 方法

### 异常处理

- 异常的层次结构 — Throwable
  - Exception
    - 运行时异常 — RuntimeException
      - IndexOutOfBoundsException
      - NegativeArgument...
      - NullPointer...
      - ClassCast...
    - IOException
      - IOEx...
      - SQL...
      - ClassNotFound...
  - Error
    - IOError
    - ThreadDeath
    - AssertionError
- 可查的异常和不可查的异常
  - 可查异常（编译器要求必须处置的异常）
  - 不可查异常(编译器不要求强制处置的异常) — 包括运行时异常（RuntimeException与其子类）和错误（Error）
- 异常关键字
  - try – 用于监听
  - catch – 用于捕获异常
  - finally – finally语句块总是会被执行
  - throw – 用于抛出异常
  - throws – 用在方法签名中，用于声明该方法可能抛出的异常
- 异常捕获处理的方法
  - try-catch
  - try-catch-finally
  - try-finally
  - try-with-resource
  - 没有异常时，catch被忽略，有异常时，try中出现异常后的语句被忽略，直接进入catch并与catch语句块逐一匹配，找到与之对应的处理程序，其他的catch语句块将不会被执行
  - finally遇见如下情况不会执行
    - 在前面的代码中用了System.exit()退出程序
    - finally语句块中发生了异常
    - 程序所在的线程死亡
    - 关闭CPU

### 内存管理

- 值传递和引用传递
  - 值传递 — 在调用函数时，将实际参数复制一份传递到函数中，这样在函数中对参数进行修改，就不会影响到原来的实际参数
  - 引用传递 — 在调用函数时，将实际参数的地址直接传递到函数中，这样在函数中对参数进行修改，就会影响到实际参数

### 垃圾回收

- 四种引用 — 强软弱虚
  - 强不会被回收
  - 软在不足时会被回收
  - 弱虚都会被回收

## Concurrent hash map

### 哈希表的原理（hashtable）

- 基本原理
  - 哈希表的核心是哈希函数和数组。数据通过哈希函数转换成一个数组索引，然后数据存储在数组的对应位置
  - 哈希函数 — 哈希表通过一个称为哈希函数的函数来计算存储数据的索引位置。理想的哈希函数应该能够将输入（通常是键）均匀分布在所有可能的哈希值中，以减少冲突（即不同的键产生相同的哈希值）。
  - 数组（哈希表本体） — 哈希函数计算出的哈希值通常用作数组的索引。数组的每一个槽位可以直接通过索引访问，这是哈希表快速访问数据的关键。
  - 处理冲突 — 当两个键的哈希值相同，即它们定位到同一个数组索引时，会发生冲突。处理冲突的方法有几种：
    - 链表法（Separate Chaining） — 每个数组槽位存储一个链表。所有映射到该索引的元素都会被存储在这个链表中。
    - 开放寻址法（Open Addressing） — 当发生冲突时，寻找数组中的下一个空槽位。常见的开放寻址技术包括线性探测、二次探测和双重哈希。
    - 双重哈希：使用第二个哈希函数确定探测的步长，以找到空槽位。
  - 动态调整大小 — 当哈希表中的元素过多时，加载因子（即哈希表中的元素个数与位置数的比值）会增加，这可能增加查找成本。 — 通常，当加载因子超过某个阈值时，哈希表会进行扩容，即创建一个更大的数组，并重新计算所有元素的位置。
- 哈希表的存和读操作？
  - 存储（插入）操作
    - 计算哈希值： — 使用哈希函数将键转换为一个整数哈希值。哈希函数设计得好的话，可以尽可能均匀地分布所有可能的键。
    - 解决冲突： — 使用如希值作为数组的索引。如果该位置已经被其他键占用（即发生冲突），根据冲突解决策略找到一个空闲位置（上面三种）
    - 存储数据： — 将键和与之关联的值存储在计算出的位置上
  - 读取（查找）操作
    - 计算哈希值： — 对要查找的键使用同一个哈希函数计算出哈希值
    - 索引定位： — 使用哈希值直接定位到数组的对应索引
    - 解决冲突： — 检查该位置的键是否与要查找的键匹配
    - 返回结果： — 如果找到匹配的键，返回对应的值 — 如果没有找到，并且已经检查了所有可能的位置（或达到搜索末尾），返回“未找到”或类似的结果
  - 动态扩容：在插入操作中，如果哈希表的加载因子超过了预定的阈值（例如 0.75），则可能会触发一个动态扩容过程。即创建一个更大的数组并重新插入所有现有元素。这样可以维持操作的高效性。
  - 哈希函数的选择：哈希函数的选择关键在于能够均匀分布键，避免过多的冲突，从而保持操作的高效性
- 底层数据结构 — 通过分段锁来控制线程安全，每个分段锁都相当于是一个小的hash table，不同的分段锁的操作可以并行执行，提高并发性。底层是一个数组+链表+红黑树的结构
- 错误检测机制：快速失败和安全失败
- 工作原理及代码实现 — ConcurrentHashMap引入了分割(Segment)的概念，只对Map的一部分(Segment)进行上锁，这样 保证带的时候，软仕的不是整个Map
- ConcurrentHashMap是如何保证线程安全的 — 使用了分段锁技术，即将哈希表分成多个段，每个段拥有一个独立的锁。这样可以在多个线程同时访问哈希表时，只需要锁住需要操作的那个段，而不是整个哈希表，从而提高了并发性能。
- ConcurrentHashMap在哪些地方做了并发控制 — 初始化桶阶段 — 如果此时段不存在，则会创建新的段。 — 为了防止多个线程都去初始化桶的问题，ConcurrentHashMap使用了CAS（Compare-And-Swap）操作来确保只有一个线程能够初始化桶。 — 等待线程初始化 — 如果另一个线程发现桶正在被初始化，它将自旋（即循环检查）直到桶初始化完成。 — 防止初始化 — 否则，当前线程将尝试初始化桶。 — put元素阶段 — 如果桶已经存在，则直接在桶上进行插入操作。 — 如果桶不存在，则会首先调用CAS插入一个新的节点到桶中。 — 扩容阶段 — 如果发现桶中有forwarding node，则表示该桶正在进行扩容。 — 在扩容期间，当前线程将帮助进行扩容操作，以减少扩容时间。 — 帮助扩容 — 当前线程将尝试将桶中的节点迁移到新的、更大的哈希表中。 — 迁移完成后，当前线程将重新尝试其原始操作。
- 为什么ConcurrentHashMap不允许null值
  - 在ConcurrentHashMap中，key和value都不允许为null
  - 在非并发的Map中（如HashMap），是可以容忍模糊性（二义性）的，而在并发Map中是无法容忍的
  - 当你拿到null时的时候，你是不知道也是因为本来就存了一个null进去还是说就是因为没找到而返回了null
  - 当我们map.get(key)返回的时候，是没办法通过map.contains(key)检查来准确的性质，因为在检测过程中可能会被其他线程所修改，而导致检测结果并不可靠，为了让ConcurrentHashMap的语义更加准确，不存在二义性的问题，他就不支持null

### 什么是红黑树

- 红黑树是一种含有红黑结点并能自平衡的二叉查找树
- 性质1：每个节点要么是黑色，要么是红色。
- 性质2：根节点是黑色。
- 性质3：每个叶子节点（NIL）是黑色。
- 性质4：每个红色结点的两个子结点一定都是黑色。
- 性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。

### 如何计算时间复杂度？

- 识别基本操作：首先，确定算法中的基本操作。基本操作是算法中执行次数与输入规模 n 最直接相关的操作。
- 计算基本操作的执行次数：分析这些基本操作随输入规模 n 的增长而执行的次数。通常，我们关注最坏情况的复杂度，即基本操作在最坏情况下的执行次数。
- 使用大O表示法：基于基本操作的执行次数，用大O表示法来描述算法的时间复杂度。大O表示法忽略常数因子和低阶项，只关注最高阶项。