Java基础

String、StringBuffer、StringBuilder的区别

	String	StringBuffer	StringBuilder
执行速度	最差	其次	最高
线程安全	线程安全	线程安全	线程不安全
使用场景	少量字符串操作	多线程环境下的大量操作	单线程环境下的大量操作

java String类是个final类，不能被继承

JDK8新特性

Lambda、Optional、Stream、Metaspace、ArrayList红黑树

1. **Lambda 表达式：** Lambda 表达式是一种更简洁的写法，可以用于表示匿名函数或闭包。它们使得在集合操作、多线程编程等方面的代码更加简洁和易读。
2. **函数式接口：** JDK 8引入了`java.util.function`包，其中包含了一些函数式接口，如`Function`、`Predicate`、`Consumer`等，用于支持使用Lambda表达式进行函数式编程。
3. **Stream API：** Stream API提供了一种新的抽象，允许以声明性的方式对集合数据进行操作，如过滤、映射、聚合等。这有助于编写更简洁和可读性更高的代码。
4. **新的日期和时间 API：** JDK 8引入了`java.time`包，提供了一套全新的日期和时间API，解决了旧的`java.util.Date`和`java.util.Calendar`等API存在的问题，使日期和时间处理更加方便和准确。
5. **默认方法和静态方法：** 在接口中，现在可以定义默认方法（default method）和静态方法（static method），这允许在不破坏实现类的情况下，为接口添加新的方法。
6. **方法引用：** 方法引用是一种更简洁的Lambda表达式写法，用于直接引用现有的方法。
7. **重复注解：** 允许在同一个地方多次使用同一种注解，提高了代码的可读性和灵活性。
8. **新的类型注解：** 引入了ElementType.TYPE_USE和ElementType.TYPE_PARAMETER两种新的注解类型，使得在更多的位置使用注解成为可能，如泛型类型参数。
9. **Nashorn JavaScript引擎：** 替代了过时的Rhino引擎，提供了更好的性能和更好的JavaScript语言支持。
10. **PermGen空间被移除：** JDK 8中的永久代（PermGen）被元空间（Metaspace）取代，用于存储类的元数据。

LTS新特性

1. **Java 8**（LTS）：
    - **Lambda 表达式和函数式接口**：引入了Lambda 表达式，简化了匿名内部类的使用，提供了函数式编程的支持。
    - **Stream API**：引入了Stream API，用于处理集合数据的函数式操作，使代码更简洁、可读性更高。
    - **新的日期和时间 API**：引入了 `java.time` 包，提供了更好的日期和时间处理方式。

- **默认方法**：接口中可以有默认实现的方法，以更好地支持接口的演化。
- **方法引用**：可以通过方法引用方式直接调用已存在的方法。
- **重复注解**：允许在同一类型上多次使用相同的注解。
- **Nashorn JavaScript 引擎**：引入了新的轻量级 JavaScript 引擎。

2. **Java 11**（LTS）：
   - **HTTP 客户端**：引入了新的 `java.net.http` 包，提供异步非阻塞的 HTTP 客户端。
   - **局部变量类型推断**：引入 `var` 关键字，使得变量的类型可以根据上下文自动推断。
   - **单元测试**：引入了单元测试框架 JEP 320，用于在模块层级上进行单元测试。
   - **动态类文件常量**：通过 `const` 关键字，可以定义在类文件中使用的常量。
   - **Epsilon 垃圾收集器**：一种不进行实际垃圾收集的垃圾收集器，用于性能测试和分析。
3. **Java 17**（LTS）：
   - **嵌套/嵌入式虚拟机（Nestmates）**：为了提高访问内部类的效率和安全性，引入了 Nestmates 功能。
   - **弱引用加强**：弱引用 API 进行了改进，提供了更好的内存管理能力。
   - **Pattern Matching for Switch**：`switch` 语句增强，支持模式匹配，可以更方便地处理多种情况。
   - **Sealed Classes**：引入密封类，限制类的继承范围，提供更好的封装性。
   - **垃圾收集器的默认选择**：在 macOS 上，G1 垃圾收集器成为默认选择。

Final和static：

final和const：const是编译时常量，final是运行时常量。

Final修饰的类不能被继承、修饰常量不能被更改

Static表示全局静态，与类有关，与实例无关

静态方法属于类，但不是类的对象

Map.entry<>遍历map;{map.getKey():map.getValue()}

Arrays排序：

Arrays.sort(arr)/Arrays.sort(arr,Collections.reverseOrder())

ArrayList排序：

arraylist.sort(Comparator. naturalOrder())/ArrayList.sort(Comparator. reverseOrder())

Queue：add（返回异常）offer（返回false）poll、element（头部、返回异常）、peek（头部元素，返回null）

Stack：add（返回false）、push（返回元素）、pop（返回元素）

Collections 提供了如下方法用于对 List 集合元素进行排序。

- void reverse(List list)：对指定 List 集合元素进行逆向排序。
- void shuffle(List list)：对 List 集合元素进行随机排序（shuffle 方法模拟了“洗牌”动作）。
- void sort(List list)：根据元素的自然顺序对指定 List 集合的元素按升序进行排序。
- void sort(List list, Comparator c)：根据指定 Comparator 产生的顺序对 List 集合元素进行排序。
- void swap(List list, int i, int j)：将指定 List 集合中的 i 处元素和 j 处元素进行交换。
- void rotate(List list, int distance)：当 distance 为正数时，将 list 集合的后 distance 个元素“整体”移到前面；当 distance 为负数时，将 list 集合的

Collection包括List（ArrayList、LinkedList）、Map（HashMap、TreeMap[SortedMap]）、Set(HashSet、TreeSet[SortedSet])、Queue（LinkedList）

Set<Map.Entry<>>set=map.entrySet()

For(Map.Entry<>me:set)me.getKey();me.getValue();

Iteator< Map.Entry<>>iter=set.iterator()

While (iter.hasNext()) me=iter.next(); me.getKey();me.getValue();

String的intern方法

1. **常量池：** 在 Java 中，有一个特殊的内存区域叫做常量池（String Pool），用于存储字符串常量。当你创建一个字符串字面值时（如 `"hello"`），它会被放入常量池。
2. **`intern()` 方法：** `String` 类的 `intern()` 方法可以将字符串对象添加到常量池中。如果字符串在常量池中不存在，就将其添加到常量池，并返回对常量池中字符串的引用。如果字符串在常量池中已经存在，就返回对已存在字符串的引用。
3. **复用字符串对象：** 通过调用 `intern()` 方法，你可以确保同样的字符串值在内存中只存在一个实例，从而节省内存和提高性能。这对于比较大量的字符串是否相等时特别有用，因为直接使用 == 比较字符串对象会比较它们的引用，而使用 `equals()` 方法比较字符串内容。

TimSort

1. **自适应长度**：Timsort 适用于不同大小的数组，它能够在小数组上使用插入排序，以避免过多的递归调用和内存占用。
2. **稳定性**：Timsort 是一种稳定排序算法，这意味着相等的元素在排序后的相对位置保持不变。
3. **分段处理**：Timsort 将待排序数组分割成一系列小块，每个块称为 "run"。首先使用插入排序对每个 run 进行排序，然后再使用归并排序合并这些 run。
4. **优化的归并步骤**：Timsort 通过使用归并排序的方式来合并 run，但是在合并的过程中采用了一些优化，比如使用二进制查找来确定合适的位置，从而提高了合并的效率。
5. **最小运行长度**：Timsort 将数组划分为多个 run，每个 run 都有一个最小长度。当遇到较小的 run 时，会使用插入排序来提高效率。
6. **最佳实践**：Timsort 在现代的编程语言中得到广泛应用，比如 Python 和 Java。在 Java 中，Arrays.sort() 方法在排序对象数组时会采用 Timsort 算法，以获得较好的性能。

Optional简介

1. `of()`：创建一个包含非空值的 Optional 实例，如果传入的值为 null，会抛出 NullPointerException。
2. `ofNullable()`：创建一个 Optional 实例，包含传入的值，但如果传入的值为 null，则创建一个空的 Optional 实例。
3. `empty()`：创建一个空的 Optional 实例，不包含任何值。
4. `isPresent()`：判断 Optional 实例是否包含非空值，如果包含返回 true，否则返回 false。
5. `ifPresent(Consumer<? super T> consumer)`：如果 Optional 实例包含非空值，则执行传入的 Consumer 操作。
6. `orElse(T other)`：如果 Optional 实例包含非空值，则返回该值，否则返回传入的默认值 other。
7. `orElseGet(Supplier<? extends T> other)`：如果 Optional 实例包含非空值，则返回该值，否则通过传入的 Supplier 生成一个默认值。
8. `orElseThrow(Supplier<? extends X> exceptionSupplier)`：如果 Optional 实例包含非空值，则返回该值，否则通过传入的 Supplier 抛出一个异常。
9. `map(Function<? super T, ? extends U> mapper)`：对 Optional 中的值进行映射转换。
10. `flatMap(Function<? super T, Optional<U>> mapper)`：对 Optional 中的值进行映射转换，并返回一个新的 Optional 对象。

String和StringBuffer优缺点

在Java中，String是一个不可变对象，这意味着每次对字符串进行操作时，都会创建一个新的String对象。而StringBuffer是一个可变的字符串类，可以在不创建新对象的情况下修改字符串。

虽然StringBuffer的结构可以提高字符串操作的效率，但是它也有一些缺点。由于StringBuffer是一个可变的字符串类，它的内部实现使用了可变大小的字符数组，这些数组的大小可以根据需要进行动态调整。这样，在进行字符串操作时，需要频繁地进行内存分配和复制，这会带来一定的开销。

相比之下，String的内部实现采用了一种称为"共享池"的机制。它会将所有的字符串字面量都存储在一个全局的字符串池中，并且对于相同的字符串字面量，只会存储一份。这样，多个字符串对象可以共享同一个字符串字面量，从而节省了内存空间。

HashMap和Hashtable的区别

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

1. HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。
2. HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。
3. 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。但这并不是一个一定发生的行为，要看JVM。这条同样也是

Enumeration和Iterator的区别。

4. 由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。
5. HashMap不能保证随着时间的推移Map中的元素次序是不变的。

Hashmap1.7和1.8的扩容机制

Java 1.7 版本中的 HashMap 扩容机制：

在 Java 1.7 中，HashMap 使用的是数组+链表的数据结构来存储键值对。扩容的主要目的是为了减少哈希冲突，提高查询性能。Java 1.7 中的扩容机制如下：

1. 当哈希表中的元素数量超过容量的 75% 时（即负载因子超过 0.75），就会触发扩容操作。
2. 扩容时，会创建一个新的数组，其容量是原数组的两倍，并且将原数组中的键值对重新计算哈希值，分布到新数组的对应位置上。

Java 1.8 版本中的 HashMap 扩容机制：

在 Java 1.8 中，HashMap 对其内部实现进行了优化，引入了红黑树来替代链表，以提高性能。Java 1.8 中的扩容机制与 Java 1.7 有一些不同之处：

1. 与 Java 1.7 相比，Java 1.8 的 HashMap 在负载因子达到 0.75 时并不总是立即触发扩容。而是会采用树化策略，即当一个桶中的链表长度达到阈值（默认为 8），将链表转换为红黑树，以提高查找效率。
2. 当红黑树的节点数量少于 6 时，会将红黑树还原为链表结构，以节省空间和维护成本。
3. 扩容时，与 Java 1.7 类似，会创建一个新的数组，容量是原数组的两倍。但在分布元素到新数组的过程中，Java 1.8 使用了更加高效的算法，减少了重新计算哈希值的次数。

fail-fast迭代器和enumratore迭代器

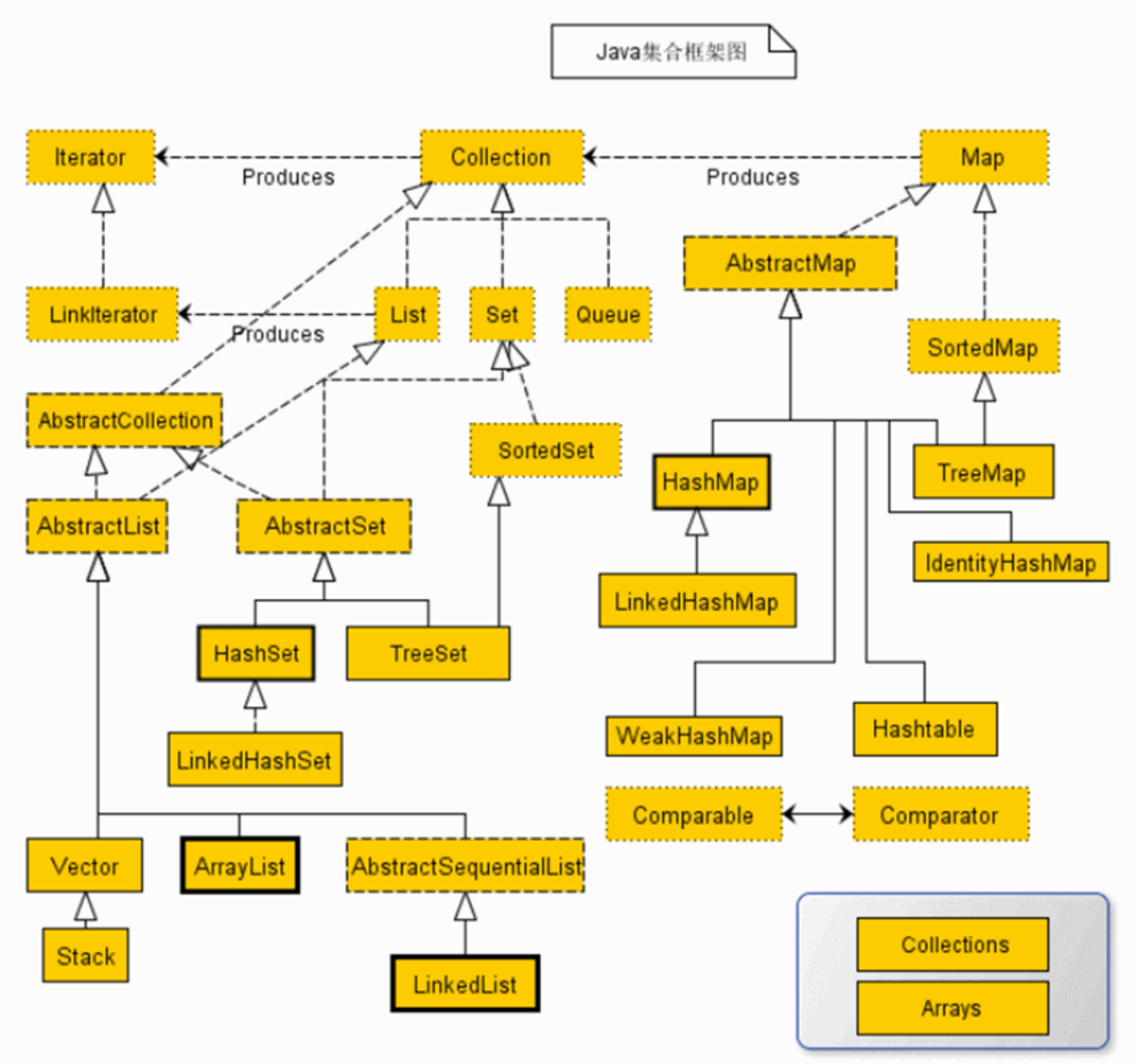
Fail-Fast 迭代器： "Fail-Fast" 是一种迭代器设计模式，主要用于检测并快速响应集合在迭代过程中的结构性修改。如果在迭代过程中，集合的结构被修改（比如增加、删除元素），则会立即抛出 ConcurrentModificationException 或类似的异常，以避免出现不一致的状态。

Fail-Fast 迭代器追求的是在出现潜在问题时尽早发现，以确保程序的正确性。Java 中的 `ArrayList`、`HashSet` 等集合类的默认迭代器就是 Fail-Fast 迭代器。

Enumerator 迭代器： Enumerator 是早期 Java 集合框架中的一种迭代器。它提供了基本的遍历功能，但不支持在遍历过程中对集合进行修改。它没有内建的检查机制来识别结构性修改，因此在迭代过程中对集合进行修改可能会导致未定义的行为。

Enumerator 迭代器相对较简单，不具备 Fail-Fast 的结构性修改检查能力。在现代的 Java 集合框架中，推荐使用 Iterator 接口来进行集合的迭代，因为 Iterator 支持更多的操作，并且可以通过 Collection 的 remove 方法来避免并发修改问题。

Fail-Safe迭代器：遍历的同时也可以修改，原因是读写分离



HashMap和HashTable的区别：

1. HashTable线程同步，HashMap非线程同步。
2. HashTable不允许<键,值>有空值，HashMap允许<键,值>有空值。
3. HashTable使用Enumeration，HashMap使用Iterator。
4. HashTable中hash数组的默认大小是11，增加方式的old*2+1，HashMap中hash数组的默认大小是16，增长方式是2的指数倍。
5. hashtable继承与Dictionary类，hashmap继承自AbstractMap类

HashTable更均匀 HashMap更高效

Synchronized：同步锁

Jdbc：

Connection：用于创建数据库连接

statement：用于执行sql语句

Java面向对象编程OOP

三大特点：封装、多态（重写、继承）、继承

封装、继承：增加代码的复用性 多态：灵活性、健壮性、可移植性

Java类的静态方法可以直接和类一起调用、非静态方法只能实例化调用

Java引用类型（强软弱虚（垃圾回收方式不同））

System.gc()java 自动回收

强引用：只要有对象指着就不会被回收

软引用：SoftReference<>空间够就分配，不够就被释放（cache）。

弱引用：WeakReference<>管你空间够不够用，直接释放

虚引用：任何时候都会被回收 主要用来跟踪对象呗垃圾回收的活动

ThreadLocal：线程隔离　线程私有的容器 应用：spring里面与数据库连接池建立connection操作 @transactional

B+树

中间节点存储的是指针，指向子节点的最大值或者最小值，叶节点存储具体的值

红黑树

红节点的子节点只能为黑节点，每个点到叶子结点的黑色节点树相同

应用：TreeMap、TreeSet、HashMap

Hashmap底层原理

数组：查询快、插入删除慢

链表：查询慢、插入删除快

1.7 数组+链表 1.8数组 + (链表 | 红黑树) 防DOS攻击

链表超过cap*factor（加载因子）数组扩容，扩容至64超8树化

原因：红黑树自平衡，到底层的距离都相同，相比AVL树旋转次数较少，每个分支的开销都是一样的

红黑树 红节点的子节点都是黑节点，叶子结点都是黑的（可为空）

多线程会出现：扩容死链（1.7）；数据错乱（1.7，1.8）

因为头插会造成依赖循环；

① HashMap 是懒惰创建数组的，首次使用才创建数组
② 计算索引（桶下标）
③ 如果桶下标还没人占用，创建 Node 占位返回
④ 如果桶下标已经有人占用
  ① 已经是 TreeNode 走红黑树的添加或更新逻辑
  ② 是普通 Node，走链表的添加或更新逻辑，如果链表长度超过树化阈值，走树化逻辑
⑤ 返回前检查容量是否超过阈值，一旦超过进行扩容
⑥ 不同
  ① 链表插入节点时，1.7 是头插法，1.8 是尾插法
  ② 1.7 是大于等于阈值且没有空位时才扩容，而 1.8 是大于阈值就扩容
  ③ 1.8 在扩容计算 Node 索引时，会优化

介绍一下 put 方法流程，1.7 与 1.8 有何不同？

① HashMap 是懒惰创建数组的，首次使用才创建数组
② 计算索引（桶下标）
③ 如果桶下标还没人占用，创建 Node 占位返回
④ 如果桶下标已经有人占用
  ① 已经是 TreeNode 走红黑树的添加或更新逻辑
  ② 是普通 Node，走链表的添加或更新逻辑，如果链表长度超过树化阈值，走树化逻辑
⑤ 返回前检查容量是否超过阈值，一旦超过进行扩容

ArrayList初始容量为0，添加一个后变成10，之后扩容均为1.5倍 n + n>>1

addAll()扩容时选max{下次扩容，源list + 加添加的长度}FailFast不允许并发修改，即遍历时不能更改元素

FailSafe牺牲一致性可以遍历完

ArrayList 是 fail-fast 的典型代表，遍历的同时不能修改，尽快失败

CopyOnWriteArrayList 是 fail-safe 的典型代表，遍历的同时可以修改，原理是读写分离

TreeMap底层原理

1. **红黑树：** 红黑树是一种自平衡的二叉搜索树，它具有以下特性：
    - 每个节点要么是红色，要么是黑色。
    - 根节点是黑色。
    - 所有叶子节点（NIL 节点）都是黑色。
    - 任何红色节点的两个子节点都是黑色。
    - 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。
2. **存储和排序：** `TreeMap` 使用红黑树来存储键值对，并且根据键的顺序进行排序。树中的每个节点包含一个键值对。通过对键进行比较，树结构使得查找、插入和删除等操作都能在 O(log n) 的时间内完成。
3. **插入操作：** 在插入一个新键值对时，`TreeMap` 会根据键的比较结果找到合适的位置，然后创建一个节点插入到树中。插入后，树可能会破坏红黑树的平衡，因此需要通过旋转和重新着色等操作来维持平衡性。
4. **查找操作：** 在查找键对应的值时，`TreeMap` 会通过比较键的大小在红黑树中进行搜索。这是一种二分查找的过程，因为红黑树是有序的。
5. **删除操作：** 在删除一个键值对时，`TreeMap` 会根据键的比较结果找到对应的节点，并根据情况进行删除操作。删除后，为了维持红黑树的平衡，可能需要进行旋转和重新着色等操作。

ArrayList和LinkedList

动态数组——双向链表、查询快，插入删除慢——查询慢，插入删除快、全局操作——头部尾部操作

LinkedList只有头部插入快，其它均慢于ArrayList

Integer 8+4+4 包含锁信息和GCroot+方法区类对象+int

Serializable：序列化接口

Volatile：解决线程安全的可见性和有序性

单例模式：单例对象不能重复创建，在默认构造方法中实力不为空时抛出异常

破坏单例的方法：反射破坏、反序列化破坏、unsafe破坏

一般在JDK中使用：Runtime、Collections

1. 饿汉式：提前创建实例
2. 枚举饿汉式：枚举是一个特殊的类，枚举的变量是类中的公有静态变量
3. 懒汉式：在getInstance()时才创建对象，且加锁synchronized实现单例，放置其他线程调用
4. DCL（双检索）懒汉式：双次检查是否创建实例，防止多次创建，且只在开始时加锁
5. 内部类懒汉式

java栈溢出

形成原因：

1. **方法调用层级过多**：当方法调用的层级过多时，会导致栈空间不足，从而出现栈溢出。这种情况通常是由递归调用或者循环调用造成的。
2. **局部变量过多**：当方法中定义的局部变量过多时，会占用大量的栈空间，从而导致栈溢出。这种情况通常是由复杂的算法或者方法嵌套造成的。
3. **线程过多**：当系统中创建的线程过多时，每个线程都会占用一定的栈空间，从而导致栈空间不足，出现栈溢出。
4. **过多的方法参数**：当方法的参数过多时，会占用大量的栈空间，从而导致栈溢出。

解决方法：

1. **优化递归算法**：尽量避免使用过深的递归算法，可以通过循环或者迭代等方式来实现。
2. **增大方法调用栈的大小**：可以通过设置JVM参数-Xss来增大方法调用栈的大小，但是需要注意不要设置过大，否则可能会影响系统的性能。
3. **减少方法调用层级**：可以通过减少方法调用层级来避免StackOverflowError异常，比如可以将多个方法合并为一个方法，避免过多的方法调用。

Error和Exception

Exception（异常）：

- Exception是指程序在运行时遇到的可处理的异常情况。
- Exception可以分为两类：可检查异常（checked exceptions）和运行时异常（runtime exceptions）。可检查异常是指需要在代码中显式捕获或声明的异常，如IOException、SQLException等。运行时异常是指不需要强制捕获的异常，如NullPointerException、ArrayIndexOutOfBoundsException等。
- 可检查异常必须在代码中使用try-catch块或者在方法签名中声明throws来处理，以确保在出现异常时能够进行适当的处理。

Error（错误）：

- Error是指程序运行时遇到的不可恢复的错误情况，通常是系统级别的问题，无法通过代码的处理来修复。
- Error不应该被程序员捕获或处理，而是由Java虚拟机（JVM）来处理。例如，OutOfMemoryError表示内存不足，StackOverflowError表示栈溢出等。
- Error通常表示应用程序或系统出现了严重问题，无法继续正常运行。