

# JavaSE

---

## Java基础

---

### Java的基本数据类型 (中)

Java 中有 8 种基本数据类型，分别为：

- 6 种数字类型：
  - 4 种整数型：`byte`、`short`、`int`、`long`
  - 2 种浮点型：`float`、`double`
- 1 种字符类型：`char`
- 1 种布尔型：`boolean`。

这八种基本类型都有对应的包装类分别为：`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`、`Character`、`Boolean`

`int`占4个字节, `char`占2个字节, `float`占4个字节, `double`占8个字节, `byte`占1个字节

java定义了`boolean`数据类型，在编译之后都使用Java 虚拟机中的`int`数据类型来代替. 所以`boolean`类型占4个字节

### 为什么用BigDecimal不用double/double计算出现什么问题? (中)

- `double`会出现精度丢失的问题
  - 计算机无法精确地表示小数, 所以做浮点数计算时会出现精度丢失问题.
- `BigDecimal`底层是用字符串存储数字, 运算也是用字符串做加减乘除计算的, 所以它能做到精确计算.

- 所以一般牵扯到金钱等精确计算，都使用Decimal。

## 基本类型和包装类型的区别？（中）

- 包装类型不赋值就是 `null`，而基本类型有默认值且不是 `null`。
- 包装类型可用于泛型，而基本类型不可以。
- 基本数据类型存放在栈中。包装类型属于对象类型，几乎所有对象实例都存在于堆中。
- 相比于对象类型，基本数据类型占用的空间非常小。

## 自动装箱与拆箱（中）

- **装箱**：将基本类型用它们对应的引用类型包装起来；调用了包装类的 `valueOf()` 方法
- **拆箱**：将包装类型转换为基本数据类型；调用了 `xxxValue()` 方法

## Integer的缓存问题？（低）

```
1 Integer i1 = 100;
2 Integer i2 = 100;
3 System.out.println(i1 == i2); // true
4
5 Integer i3 = 1000;
6 Integer i4 = 1000;
7 System.out.println(i3 == i4); // false
```

为什么出现上面这种奇怪的现象？

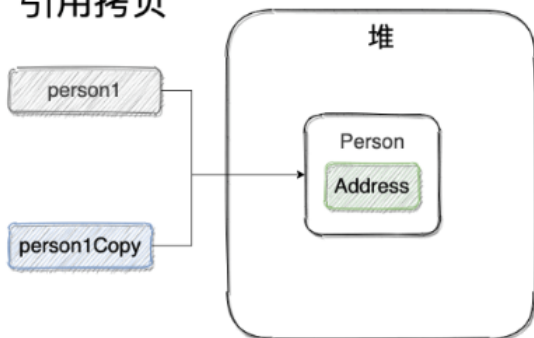
- Java的Integer类内部实现了一个静态缓存池，用于存储特定范围内的整数值对应的Integer对象。
- 默认情况下，这个范围是-128至127。当创建一个在这个范围内的整数对象时，并不会每次都生成新的对象实例，而是复用缓存中的现有对象，会直接从内存中取出，不需要新建一个对象。

所以, 在对比是一定要用equals()。

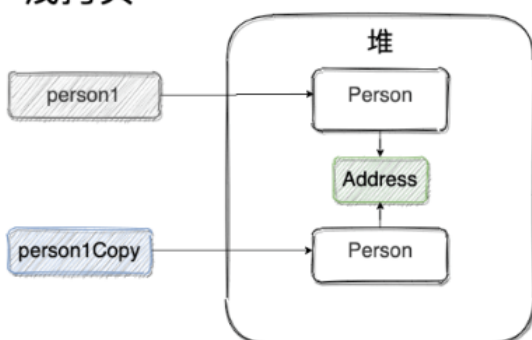
## 深拷贝和浅拷贝区别？什么是引用拷贝 (中)

- **浅拷贝**：浅拷贝会在堆上创建一个新的对象（区别于引用拷贝的一点），不过，如果原对象内部的属性是引用类型的话，浅拷贝会直接复制内部对象的引用地址，也就是说拷贝对象和原对象共用同一个内部对象。
- **深拷贝**：深拷贝会完全复制整个对象，包括这个对象所包含的内部对象。
- **引用拷贝**：引用拷贝就是两个不同的引用指向同一个对象。

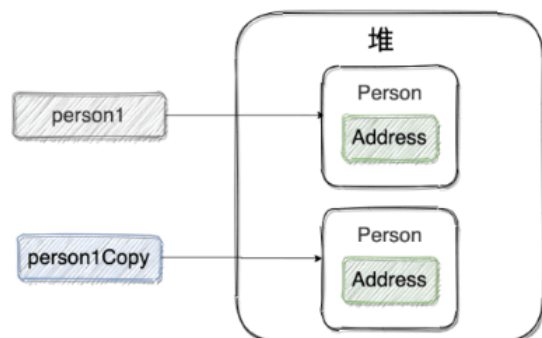
引用拷贝



浅拷贝



深拷贝



## == 和 equals() 的区别 (高)

== 对于基本类型和引用类型的作用效果是不同的：

- 对于基本数据类型来说，== 比较的是值
- 对于引用数据类型来说，== 比较的是对象的内存地址

equals() 方法存在两种使用情况：

- **类没有重写 equals() 方法**：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，使用的默认是 Object 类 equals() 方法。
- **类重写了 equals() 方法**：一般我们都重写 equals() 方法来比较两个对象中的属性是否相等；若它们的属性相等，则返回 true(即，认为这两个对象相等)。

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。

## hashCode() 有什么用？ (高)

hashCode() 的作用是获取哈希值。这个哈希值的作用是**确定该对象在哈希表中的索引位置（可以快速找到所需要的对象）**

Java 用 hashCode() 和 equals() 判断是否为同一个对象

- 如果两个对象的 hashCode 值相等，那这两个对象不一定相等（哈希碰撞）。
- 如果两个对象的 hashCode 值相等并且 equals() 方法也返回 true，我们才认为这两个对象相等

- 如果两个对象的 `hashCode` 值不相等，我们就可以直接认为这两个对象不相等。

## 重写equals为什么要重写hashcode? (高)

因为java判断两个对象是否是相等的, 需要先比较hashcode是否一致, 如果hashcode不一致那么就认为不相等.

如果没有重写hashcode, 那么两个相等的对象由于hashcode不相等, 就会被认为是不相等的. 但是按照重写的equals规则, 他们应该是相等的.

在集合中, 如set集合去重中就会出现, 两个相等的对象放到set中都可以存在的怪象.

## 抽象类和接口的区别 (中)

- 抽象类和接口都不能直接实例化。如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。
- 抽象类要被子类继承，接口要被类实现。
- 接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现。
- 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。
- 抽象方法要被实现，所以不能是静态的，也不能是私有的
- 抽象类是对事物的抽象，即对类抽象；接口是对行为抽象，即局部抽象。抽象类对整体形为进行抽象，包括形为和属性。接口只对行为进行抽象。

- 抽象类是多个子类的父类，定义了子类大概的共性的东西，是一种模板式设计；接口是一种行为规范，是一种辐射式设计。

## 面向对象的三大特征 (中)

- 封装
  - 为了提高代码的安全性，隐藏对象的内部细节，封装将对象的内部状态（字段、属性）隐藏起来，并通过定义公共的方法（接口）来操作对象
  - 外部代码只需要知道如何使用这些方法而无需了解内部实现
- 继承
  - 允许一个类（子类）继承另一个类（父类）的属性和方法的机制
  - 子类可以重用父类的代码，并且可以通过添加新的方法或修改（重写）已有的方法来扩展或改进功能
  - 提高了代码的可重用性和可扩展性
- 多态
  - 多态是指相同的操作或方法可以在不同的对象上产生不同的行为，通过方法的重载和重写实现
  - 多态允许以一致的方式处理不同类型的对象，提高了代码的灵活性

## 面向对象和面向过程的区别 (中)

面向过程：直接将解决问题的步骤分析出来，然后用函数把步骤一步一步实现，然后再依次调用就可以了。面向过程思想偏向于我们做一件事的流程，首先做什么，其次做什么，最后做什么。

面向对象: 将构成问题的事物, 分解成若干个对象, 建立对象的目的不是为了完成一个步骤, 而是为了描述某个事物在解决问题过程中的行为。 需要完成什么事情, 直接让某个对象来干即可。

## 类和对象 (中)

类: 像是一个抽象的设计图/模板. 类往往保存一类事物的共性(属性), 共有行为.

对象: 是通过这个设计图/模板创造出来具体实例. 实例往往是共性个性化的表现.

## 说一说你对多态的理解 (中)

- 子类其实是一种特殊的父类, 因此Java允许把一个子类对象直接赋给一个父类引用变量, 无须任何类型转换, 或者被称为向上转型, 向上转型由系统自动完成。
- 当把一个子类对象直接赋给父类引用变量时, 例如 `FatherObj o = new SonObj()`, 这个编译时类型是 `FatherObj`, 而运行时类型是 `SonObj`, 当运行时调其方法时, 其方法行为实际是子类的行为, 也就是 `SonObj` 的行为。
- 这就可能出现: 相同类型的变量、调用同一个方法时出现不同的行为, 这就是所谓的多态

## 方法的重载和重写有什么区别 (中)

1. 重载方法法重载指的是在同一个类中, 方法名相同但参数列表不同

2. 重写是在子类中重新定义父类中已有的方法，方法名和参数列表必须相同

## 静态变量和静态方法与非静态有什么区别? (中)

静态变量和静态方法是与类本身关联的，而不是与类对象关联。它们在内存中只存在一份，可以被类的所有实例共享。

换句话说，静态是属于类的，是被类所有对象共享的。

### 静态变量

- 共享性：所有该类的实例共享同一个静态变量。如果一个实例修改了静态变量的值，其他实例也会看到这个更改。
- 初始化：静态变量在类被加载时初始化，只会对其进行一次分配内存。
- 访问方式：静态变量可以直接通过类名访问，也可以通过实例访问，但推荐使用类名访问。

### 静态方法

- 共享性：所有该类的实例共享同一个静态方法。
- 访问方式：静态方法可以直接通过类名调用，不需要创建对象。
- 访问静态成员：静态方法可以直接调用其他静态变量和静态方法，但不能直接访问非静态成员。因为静态没有依赖具体对象。

## final的作用 (中)

- 被final修饰的类无法继承
- 被final修饰的方法无法重写
- 被final修饰的变量为常量，无法重新赋值



- final修饰基本类型变量, 无法修改
- final修饰引用类型, 这个引用无法指向其他对象, 也就是地址无法修改, 但是对象属性可修改.

# String

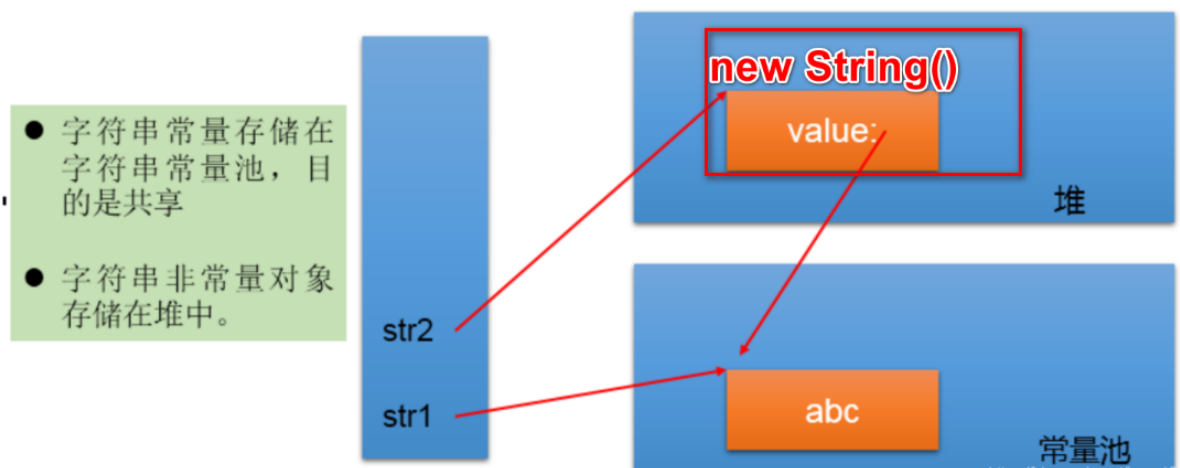
## String的不可变性 (中)

String类中包含一个数组, 储存数组的每一个字符: `private final byte[] value;`

- final数组, 地址不能改变, 导致长度不能改变
- private, 数组中的内容不能改变

## String直接赋值和new对象的区别 (中)

String str1 = "abc";与String str2 = new String("abc");的区别?



```
1 String s1 = "abc";
2 String s2 = new String("abc");
3
4 System.out.println(s1 == s2); //false
5 System.out.println(s1.equals(s2)); //true
```

String重写了equals, 所以它的equals是先比较对象的地址, 再比较里面的值.

## String s1 = new String("abc");这句话创建了几个字符串对象? (中)

会创建 1 或 2 个字符串。

- 如果字符串常量池中不存在字符串对象“abc”的引用，那么会在堆中创建2个对象
  - 一个是new的String对象
  - 一个是char[]对应的常量池中的数据: "abc"
- 如果字符串常量池中不存在字符串对象“abc”的引用，那么会在堆中创建1个对象, 就是new的String对象.

## String和StringBuffer和StringBuilder区别 (高)

- String: 字符串变量, private final修饰, 不可变!
- StringBuffer: 字符串变量 (线程安全,可变) 没有使用 `final` 和 `private` 关键字修饰
- StringBuilder: 字符串变量 (线程不安全,可变) 没有使用 `final` 和 `private` 关键字修饰
- StringBuilder是StringBuffer的简易版, 更快!

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` 或 `StringBuilder` 每次都会对 `StringBuffer` 或 `StringBuilder` 对象本身进行操作, 而不是生成新的对象并改变对象引用。

**对于三者使用的总结:**

1. 操作少量的数据: 适用 `String`
2. 单线程做大量字符串拼接操作: 适用 `StringBuilder`
3. 多线程做大量字符串拼接操作: 适用 `StringBuffer`

## 字符串拼接用“+” 还是 `StringBuilder`? (中)

对象引用和“+”的字符串拼接方式，实际上是通过 `StringBuilder` 调用 `append()` 方法实现的，拼接完成之后调用 `toString()` 得到一个 `String` 对象。

不过，在循环内使用“+”进行字符串的拼接的话，存在比较明显的缺陷：**编译器不会创建单个 `StringBuilder` 以复用，会导致创建过多的 `StringBuilder` 对象。** `StringBuilder` 对象是在循环内部被创建的，这意味着每循环一次就会创建一个 `StringBuilder` 对象。

所以要把 `new StringBuilder()` 放在循环外部。

## 字符串常量池的作用了解吗? (中)

**字符串常量池** 是 JVM 为了提升性能和减少内存消耗针对字符串 (`String` 类) 专门开辟的一块区域，主要目的是为了避免字符串的重复创建

开发中使用的字符串很可能有大量的重复，字符串常量池就可以将重复的字符串只保存一份，极大地节省了内存。

# 集合

---

## 集合的使用场景与实现 (高)

存取有序用List

- ArrayList： 基于动态数组实现。底层使用数组作为存储结构, 会对数组动态扩容.
- LinkedList： 基于双向链表实现。
- Vector： 线程安全版ArrayList, 底层用数组实现, 但是大量使用synchronized加锁, 性能差, 已经不会使用了

希望自动去掉重复元素用Set

- HashSet： 基于HashMap实现。底层使用一个数组和链表/红黑树的结构来存储元素, 无序无重复
- TreeSet： 基于TreeMap实现。因为TreeMap底层是红黑树, 而红黑树是一种排序树, 故可以自动排序.

通过唯一的key找value用map

- HashMap： 基于哈希表实现。底层使用一个数组+链表/红黑树的结构来存储键值对。key无序无重复.
- TreeMap： 基于红黑树实现, 因为红黑树是一种排序树, 故可以对key自动排序.
- Hashtable： HashMap的线程安全版, 但是大量使用synchronized加锁, 性能差, 已经不会使用了
- ConcurrentHashMap: HashMap的线程安全版, 没有大量加锁, jdk1.7使用分段锁设计, 1.8开始锁桶, 性能比Hashtable好很多, 经常使用.

## 队列先进先出用Queue

- LinkedList: 他也实现了Queue接口, 可以先进先出
- ArrayDeque: 基于动态数组的双端队列。底层使用循环数组实现.

## 栈后进先出用Stack

- Stack: 继承Vector, 大量使用synchronized加锁, 性能差, 已经不会使用了.
- ArrayDeque: 基于动态数组的双端队列。底层使用循环数组实现. 因为是双端队列, 故可以当栈用.

# ArrayList (高)

## ArrayList 与 LinkedList 的区别?

### 内部数据结构

- ArrayList: 基于动态数组实现
- LinkedList: 基于双向链表实现

### 遍历性能

- ArrayList: 支持快速的随机访问和遍历, 因为可以直接通过索引访问元素
- LinkedList: 随机访问性能较差, 因为必须从链表的头部或尾部开始遍历, 直到达到目标索引

### 插入和删除

- ArrayList: 在末尾进行插入和删除操作是高效的,但在中间或开头插入和删除需要移动元素,性能较差
- LinkedList: 插入和删除元素的性能相对较好,特别是在链表中间或头尾插入和删除元素时.

内存空间占用:

- ArrayList的空间浪费主要体现在底层数组需要为新元素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 (因为链表要存放后继和前驱节点)

使用场景

- ArrayList: 适用于需要频繁随机访问元素,而对插入和删除操作要求不那么严格的场景
- LinkedList: 适用于需要频繁插入和删除操作,而对随机访问的需求较少的场景

以上是常规对比,也可以理解为是动态数组和链表的对比.

但是java的LinkedList有严重缺陷,插入删除的性能其实很多场景比ArrayList要差. 为什么?

- 因为Java的LinkedList实现了List接口,如果往某个位置插入数据,需要从链表头部查找到对应的节点,然后再插入. 找节点的时间复杂度是 $O(n)$ ,插入是 $O(1)$ . 我们讨论链表插入性能,往往讨论的是拿到某节点后的插入性能,但其实找节点也需要耗时,所以Java的LinkedList插入性能很差. 只有往链表头部或者尾部插入性能会比较好.
- 原因之二是因为数组虽然插入需要移动元素,但是由于CPU有三级缓存,数组又是连续空间,所以很容易就触发缓存,缓存的操作速度又非常快. 所以这就导致,即便要移动数据,在缓存中移动数据未必比链表插入要慢. 所以开发使用99%的场景都应该用ArrayList.

## ArrayList的扩容机制

1. 当调用ArrayList的无参构造new对象时, ArrayList对象的初始容量为0.
2. 当插入一个元素时, ArrayList会进行首次扩容(无参构造才会用首次扩容). 首次扩容, ArrayList会创建一个长度为10的新数组, 替换掉原来的旧数组. 此时ArrayList的长度就为10.
3. 当插入10个数据后, 要插入第11个数据时, 容量不足, 会触发第2次扩容. 第二次扩容会扩容原容量的1.5倍, 之后扩容都是原容量的1.5倍.
  - ArrayList每次扩容都会创建新数组, 然后把数据转移到新数组中.

## HashMap (高)

### HashMap和Hashtabe的区别

只关注一点就行了, 就是是否线程安全. 其他不用记!

- HashMap线程不安全, Hashtabe线程安全
- Hashtabe的线程安全是大量使用synchronized加锁, 性能差, 已经不会使用了
- 并发环境下建议使用ConcurrentHashMap, 它的性能更好.
  - 结合ConcurrentHashMap那一节的内容回答即可.

## HashMap底层数据结构, 1.7和1.8有何不同?

- 1.7的HashMap是哈希表, 数组 + 链表.
- 1.8的HashMap是数组 + (链表 或 红黑树)

## HashMap什么时候进行扩容

- `HashMap` 默认的初始化大小为16, 默认负载因子为0.75.
- 当hashmap中的元素个数超过**数组大小\*负载因子(loadFactor)**时, 就会进行数组扩容.
- 之后每次扩充, 容量变为原来的 2 倍

## HashMap为什么要使用红黑树, 为啥不用平衡二叉树

- 当某个位置, Hash冲突严重, 则链表的长度会很长. 那么查找的时候依次比较, 效率会很低  $O(n)$ .
- 将链表转为红黑树, 因为红黑树是排序树, 查找效率一般是  $O(\log N)$ , 使用红黑树就提高了查找效率.
- 平衡二叉树追求绝对平衡, 每次插入新节点之后需要旋转的次数不能预知, 自平衡效率低
- 红黑树放弃了追求完全平衡, 追求大致平衡, 在与平衡二叉树的时间复杂度相差不大的情况下, 自平衡的效率更高

## 什么时候会树化

- 要满足两个条件
  - 链表长度超过树化阈值8
  - 数组长度大于等于64



- 当链表长度超过8时, 若数组长度小于64, 则会对数组进行扩容, 然后二次哈希的值就会变, 此时链表的部分值就会重新分配到数组其他位置. 二次分配后链表长度未必大于8, 所以必须同时满足两个条件才会树化.

## 为何一上来不树化

- 刚开始链表的长度可能只有三四个, 如果此时树化, 那么树化后查询的效率和短链表差不多. 所以, 在短链表的情况下, 树化意义不大.
- 而且链表的节点是Node, 红黑树的节点为TreeNode. TreeNode的内存占用大于Node. 所以非必要不树化.

## 为何树化阈值为8

- 红黑树是为了防止链表超长时性能下降, 树化应当是偶然情况. 长度超过8的链表出现几率非常小, 选择8就是为了让树化几率足够小
- hash表的查找, 更新的时间复杂度是 $O(1)$ . 而红黑树的查找, 更新的时间复杂度是 $O(\log_2 n)$ , TreeNode占用空间也比普通Node的大, 所以如非必要, 尽量还是使用链表。

## 索引如何计算? hashCode有了, 为何还有hash()方法? 数组容量为何是 $2^n$ ?

- 索引计算方式
  - 对任何一个对象调用其hashCode()方法会获得其原始hash值.

- 对原始hash值再调用HashMap的hash方法进行二次hash, 获取到二次hash值.
- 二次hash值对数组容量进行取余操作获取到存放的数组下标.
- 为何需要二次hash?
  - 二次hash是为了让hash值分布更加均匀, 减少hash冲突, 从而使链表更短, 因此也就提升了查找效率.
- 数组容量为何是 $2^n$ ?
  - 计算索引时, 如果是2的n次方可以使用位与运算代替取模运算, 效率更高
  - 数组容量为质数会使hash值分布均匀, 但是 $2^n$ 计算索引的效率更高.

## HashMap的put()方法流程

### 1.8的put流程

1. HashMap 是懒惰创建数组的, 首次使用才创建数组
2. 调用hashCode, 然后再调用hash(), 二次hash来计算索引 (桶下标)
3. 如果桶下标还没人占用, 创建Node放入数据后返回
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.
  1. 扩容时, 先将新的数据放进数组, 然后创建新的数组, 再将旧数组元素迁移到新数组

### 1.8和1.7的put流程不同之处

- 链表插入节点时, 1.7是头插法, 1.8是尾插法
- 1.8有判断链表长度, 树化的逻辑. 1.7没有.

## 负载因子为何是0.75

- **0.75是在占用空间和查询时间中取得了比较好的平衡. (扩容阈值=数组大小\*负载因子)**
- 大于0.75, 冲突增加了, 数组空间就节省了, 但是链表就会比较长, 影响性能.
  - 若负载因子为1, 则 $16*1 = 16$ , 只有当存满16个元素之后, 才会扩容. 节省空间, 影响性能.
- 小于0.75, 冲突减少了, 链表会比较短, 数组扩容会很频繁, 空间占用增多.
  - 若负载因子为0.25, 则 $16*0.25=4$ , 当元素个数大于4个, 就会扩容. 浪费空间, 提高性能

## 多线程操作HashMap会出现什么问题

知道hashMap是线程不安全即可. 具体并发性会出现什么问题了解即可.

- 扩容死链 (1.7)
  - 扩容的时候线程切换, 两个线程都要进行扩容.
  - 因为1.7是头插法, 进行数组扩容的时候, 需要链表迁移. 在并发环境下, 会出现循环链表, 造成扩容死链问题.
- 数据错乱 (1.7, 1.8)

两个线程都要放入一个新数据 (两个数据索引一致, 且该索引下无链表)

当有两个线程都判断为null, 且都进入if逻辑  
那么后一个去newNode, 就会覆盖前一个

```
if ((p = tab[i = (n - 1) & hash]) == null)
    tab[i] = newNode(hash, key, value, next: null);
```

## 重写HashMap的equal方法不当会出现什么问题?

HashMap在比较元素时, 会先通过hashCode进行比较, 相同的情况下再通过equals进行比较。

- 所以equals相等的两个对象, hashCode一定相等
- hashCode相等的两个对象, equals不一定相等 (比如hash冲突的情况)

重写了equals方法, 不重写hashCode方法时, 可能会出现equals方法返回为true, 而hashCode方法却返回false. 这会导致在hashmap等类中存储多个一模一样的对象, 导致出现覆盖存储的数据的问题.

equals()和hashCode()的实现应该遵循以下规则:

- 如果 `o1.equals(o2)`, 那么 `o1.hashCode()==o2.hashCode()` 总为true
- 如果 `o1.hashCode()==o2.hashCode()`, 并不意味着 `o1.equals(o2)` 会为true

# ConcurrentHashMap (高)

## 对比

### 与HashMap的区别是什么？

ConcurrentHashMap是HashMap的升级版，HashMap是线程不安全的，而ConcurrentHashMap是线程安全。而其他功能和实现原理和HashMap类似

### 与Hashtable的区别是什么？

- Hashtable也是线程安全的，整个Hashtable对应一把锁，同一时刻，只能有一个线程操作它，并发性低
- 1.7的ConcurrentHashMap使用分段锁，也就是Segment+HashEntry数组+链表的结构，相当于将数组切分为多个Segment. 每个Segment对应一把锁，如果多个线程访问不同的Segment, 则不会冲突.
- 1.8开始 ConcurrentHashMap将链表的每个头节点或者红黑树的根节点作为锁(锁桶)，如果多个线程访问的头节点不同，则不会冲突. 也就是说，只要没有hash冲突，多个线程就可以同时访问ConcurrentHashMap.

### JDK8的ConcurrentHashMap和JDK7的ConcurrentHashMap有什么区别？

1. JDK8中新增了红黑树
2. JDK7中使用的是头插法，JDK8中使用的是尾插法
3. JDK7中使用了分段锁，而JDK8中没有使用分段锁，而是锁住链表或者红黑树的头结点. JDK 1.7 最大并发度是 Segment 的个数，默认是 16。JDK 1.8 最大并发度是数组的大小，并发度更大

4. JDK7中使用了ReentrantLock, JDK8中没有使用ReentrantLock了, 而使用了Synchronized
5. JDK7中的扩容是每个Segment内部进行扩容, 不会影响其他Segment, 而JDK8中的扩容和HashMap的扩容类似, 只不过支持了多线程扩容, 并且保证了线程安全

## 特性

### ConcurrentHashMap是如何保证并发安全的?

- JDK1.87中ConcurrentHashMap是通过ReentrantLock+CAS+分段思想来保证的并发安全的
  - ConcurrentHashMap的put方法会通过CAS的方式, 把一个Segment对象存到Segment数组中
  - 一个Segment内部存在一个HashEntry数组, 相当于分段的HashMap, Segment继承了ReentrantLock, 每段put开始会加锁。
- JDK1.8通过CAS+synchronized +锁桶的思想来保证并发安全的.
  - 它将每个桶（数组中的每个位置）作为独立的锁单位, 当操作不同的桶时, 线程无需竞争同一把锁
  - 锁的粒度更细了. 并发度更高
  - **synchronized 锁头节点**: 插入或修改数据时, 仅对当前桶的头节点加锁, 也就是只锁桶.

### JDK8中的ConcurrentHashMap为什么使用synchronized来进行加锁?

在JDK1.6中，对synchronized锁的实现引入了大量的优化，并且synchronized有多种锁状态，会从无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁一步步转换。此时synchronized性能并不弱于ReentrantLock(这是前提)

- JDK7中使用了分段锁，所以对于一个ConcurrentHashMap对象而言，分了几段就得有几个ReentrantLock对象，表示得有对应的几把锁。
- JDK8中会锁桶，当并发量比较高的时候，需要锁的桶就很多，锁的数量就会增多。
- ReentrantLock是一个对象，而synchronized是一个关键字，当桶比较多时，ReentrantLock对象也比较多，那么就会占用很多内存，故jdk1.8使用synchronized关键字来加锁就会更节省内存。

### ConcurrentHashMap是如何扩容的？

- 它在put的方法的最后一步来判断是否扩容。
- 当链表长度大于8，但是数组长度小于64就会尝试扩容，每次扩两倍。
- 在transfer方法里面会创建一个原数组的俩倍的node数组来存放原数据（扩容过程也需要用到cas方式去判断）
- 当前线程发现此时map正在扩容，则协助扩容

### ConcurrentHashMap 的 put 方法执行逻辑是什么？

JDK1.7

- 先通过key的hash判断得到Segment数组的下标，然后将这个Segment上锁，然后再次通过key的hash得到Segment里HashEntry数组的下标，然后插入头插法插入链表，下面这步其实就和HashMap一致了。

- 尝试获取锁失败肯定就有其他线程存在竞争, 所以就先尝试自旋获取锁, 如果重试的次数达到了一定程度, 则阻塞获取锁, 保证能够获取成功.

## JDK1.8

- 当塞入一个值的时候, 先计算 key 的 hash 后的下标, 通过自旋死循环保证一定可以新增成功
- 如果计算到的下标还没有 Node , 那么就通过 cas 塞入新的 Node
- 如果已经有 node 则通过 synchronized 将这个 node 上锁, 这样别的线程就无法访问这个 node 及其之后的所有节点
- 然后如果是链表就走链表的插入逻辑, 如果是红黑树就走红黑树插入逻辑. 当在链表长度达到 8 的时候, 数组扩容或者将链表转换为红黑树

## ConcurrentHashMap 的 get 方法执行逻辑是什么?

## JDK1.7

1. 根据 key 计算出 hash 值定位到具体的 Segment , 再根据 hash 值获取定位 HashEntry 对象, 并对 HashEntry 对象进行链表遍历, 找到对应元素。
2. 由于 HashEntry 涉及到的共享变量都使用 volatile 修饰, volatile 可以保证内存可见性, 所以每次获取时都是最新值。

## JDK1.8

1. 根据 key 计算出 hash 值, 判断数组是否为空;
2. 如果是首节点, 就直接返回;
3. 如果是红黑树结构, 就从红黑树里面查询;
4. 如果是链表结构, 循环遍历判断。



5. 有volatile修饰, 直接就能拿到最新值

### ConcurrentHashMap 的 get 方法是否要加锁, 为什么?

- get 方法不需要加锁。因为 Node 的元素 value 和指针 next 是用 volatile 修饰的, 在多线程环境下线程A修改节点的 value 或者新增节点的时候是对线程B可见的。
- 这也是它比其他并发集合比如 Hashtable、用 Collections.synchronizedMap()包装的 HashMap 效率高的原因之一

### ConcurrentHashMap 的并发度是什么

- 在JDK1.7中, 实际上就是ConcurrentHashMap中的分段锁个数, 即Segment[]的数组长度, 默认是16, 这个值可以在构造函数中设置。
- 在JDK1.8中, 已经摒弃了Segment的概念, 选择了Node数组+链表+红黑树结构, 并发度大小依赖于数组的大小

## 反射

---

### 反射的基本思想 (中)

反射机制是在运行时, 能够动态获取类的所有属性和方法; 动态调用对象任意方法, 动态的创建对象

但是反射需要在运行时动态解析类、方法、字段的元数据信息, 性能较差

反射特性:

- 运行时类信息访问：反射机制允许程序在运行时获取类的完整结构信息，包括类名、包名、父类、实现的接口、构造函数、方法和字段等。
- 动态对象创建：可以使用反射API动态地创建对象实例，即使在编译时不知道具体的类名。这是通过Class类的newInstance()方法或Constructor对象的newInstance()方法实现的。
- 动态方法调用：可以在运行时动态地调用对象的方法，包括私有方法。这通过Method类的invoke()方法实现，允许你传入对象实例和参数值来执行方法。
- 访问和修改字段值：反射还允许程序在运行时访问和修改对象的字段值，即使是私有的。这是通过Field类的get()和set()方法完成的。

## 你平时什么时候会用到反射（中）

- 当要从配置文件中读配置创建类对象时需要使用反射。
  - 配置文件配置某个类的全类名, 然后就需要读配置, 然后反射创建对象.
- 使用工厂模式时, 往往也需要根据全类名来获取对象, 也会使用反射创建对象.
- 当开发注解时, 往往需要反射来获取某个类/字段的注解
- 当使用spring, mybatis时, 这些框架底层会大量使用反射
  - Spring 的IoC机制：会通过反射实例化 Bean、注入依赖
  - Spring的AOP用到了动态代理, 也是大量使用反射
  - MyBatis 的 Mapper 接口动态代理：反射生成接口的代理对象，执行 SQL 映射方法

# 多线程

---

## 多线程基础

### 聊聊线程和进程（高）

进程是程序的一次执行过程，是系统运行程序的基本单位，是操作系统分配资源的最小单位。一个进程会有一个主线程

- 在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程
- 而 main 函数所在的线程就是这个进程中的一个线程，也称主线程

线程是一个比进程更小的执行单位，一个进程在其执行的过程中可以产生多个线程

- 线程共享进程的**堆和方法区**资源
- 每个线程有自己的**程序计数器、虚拟机栈和本地方法栈**
- 线程切换的代码比进程小得多，线程也被称为轻量级进程

### Java创建线程有几种方式（高）

常规回答

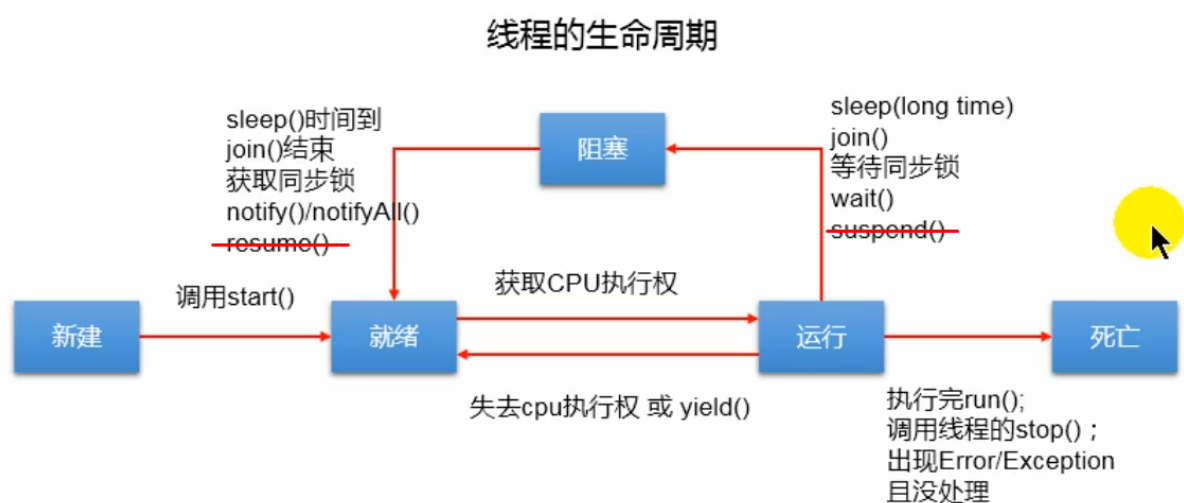
- 继承Thread类, 重写run方法
- 实现 Runnable 接口
- 实现Callable接口
- 通过线程池创建

以上回答并未触及本质, 接下来聊聊本质

- Java创建线程有且只有一种方式, 就是继承Thread类重写run方法, 调用Thread类的start方法
- 实现Runnable和Callable的还是要将实现类对象传入Thread类中, 调用Thread类的start方法. 所以本质创建线程还是依靠Thread类中的start方法.
- 实现Runnable和Callable实际上是创建了一个线程任务. 然后调用Thread类中的start方法, start方法调用start0方法, start0是一个本地方法, 由C/C++编写, 用来进行系统调用创建新线程, 然后用新线程来执行线程任务.
- 这也说明了为何实现Runnable或者Callable后调用run方法不能开启新线程, 是因为开启新线程本质上需要调用Thread的start0方法.

## 线程状态与线程生命周期 (中)

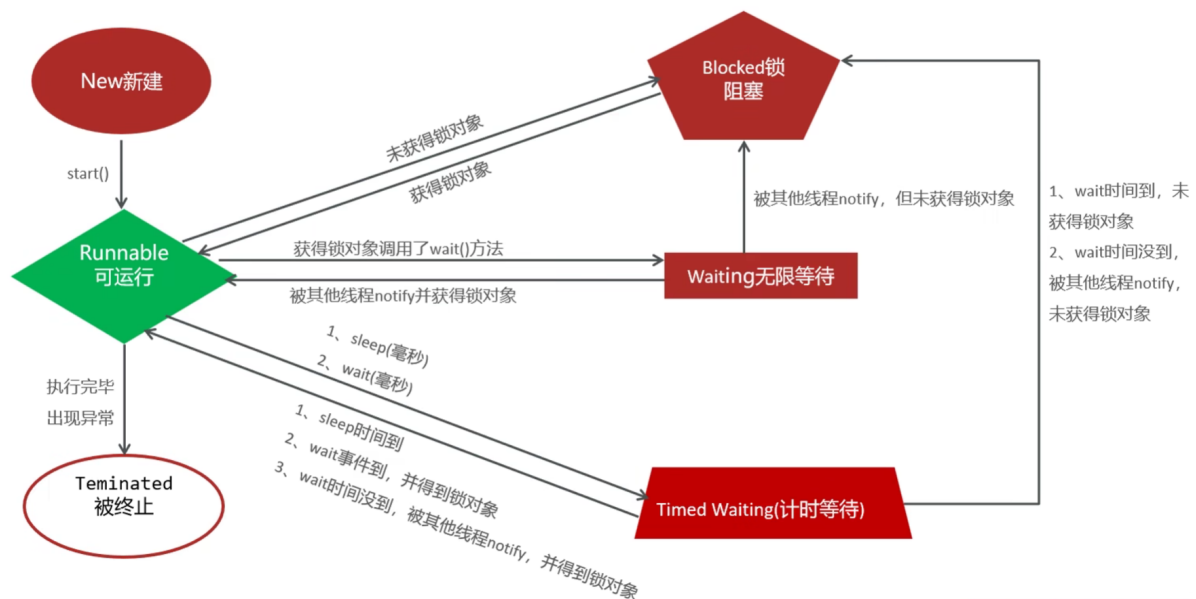
操作系统中线程的5大状态



- 线程刚开始处于新建状态
- 当线程启动时, 获取了除CPU以外的其他资源后就到了就绪态
- 当线程获取了cpu, 开始被cpu执行, 就进入运行态
- 当线程失去了CPU, 就又进入就绪态

- 当线程被阻塞, 或者等待IO等操作, 就是阻塞态.
- 当IO执行完成或者等待结束, 就进入就绪态, 等待分配CPU.
- 当线程被终止, 或者出现异常, 线程崩溃, 就是终止/死亡状态.

## java中Thread.state枚举类定义的6状态



注意: sleep()不会释放锁, 所以sleep()结束后会可以直接进入就绪态(runnable). wait()会释放锁, 所以wait()结束后要去抢锁, 抢到则进入就绪态(runnable), 没抢到则进入锁阻塞(Blocked).

线程状态	描述
NEW(新建)	线程刚被创建, 但是并未启动。
Runnable(可运行)	线程已经调用了start()等待CPU调度
Blocked(锁阻塞)	线程在执行的时候未竞争到锁对象, 则该线程进入Blocked状态; 。
Waiting(无限等待)	一个线程进入Waiting状态, 另一个线程调用notify或者notifyAll方法才能够唤醒
Timed Waiting(计时等待)	同waiting状态, 有几个方法有超时参数, 调用他们将进入Timed Waiting状态。带有超时参数的常用方法有Thread.sleep 、 Object.wait。
Teminated(被终止)	因为run方法正常退出而死亡, 或者因为没有捕获的异常终止了run方法而死亡。

## 什么是死锁 (中)

- 多线程争抢资源, 只有得到资源才能继续执行.
- 但是每个线程都持有一部分资源, 都等待对方释放资源.
- 多个线程互相僵持, 导致都无法运行

## 死锁的四个必要条件 (中)

1. **互斥条件**: 只有对**临界资源**(需要互斥访问的资源)的争夺才会产生死锁
2. **不可剥夺条件**: 进程在所获得的资源未释放前, **不能被其他进程强行夺走, 只能自己释放。**
3. **请求保持条件**: 两个进程各占有一部分资源, 保持**占有一部分资源的同时都请求对方让出另一部分资源**
4. **循环等待条件**: 双方都**等待对面让出资源, 产生僵持**

## 死锁检测, 预防, 避免 (中)

### 检测死锁

- `jps` 可以查看定位进程号, `jstack 进程号` 可以查看栈信息, 来排查死锁
- `jconsole` 可以用来检测死锁
- arthas这种工具也可以用来检测排查死锁

### 预防死锁

- **破坏互斥条件**: Java的ThreadLocal, 每个线程都拥有自己数据副本, 自己访问自己的, 不需要互斥访问.
- **破坏请求与保持条件**: 一次性申请所有的资源

- **破坏不可剥夺条件**：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源
  - 超时放弃 (**破坏不可剥夺条件**)
    - Lock接口提供了boolean tryLock(long time, TimeUnit unit) 方法, 如果一定时间没获取到锁就放弃.
- **破坏循环等待条件**：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件
  - 指定获取锁的顺序 (**破坏循环等待条件**)
    - 比如某个线程只有获得A锁和B锁才能对某资源进行操作.
    - 规定获取锁的顺序，比如只有获得A锁的线程才有资格获取B锁，按顺序获取锁就可以避免死锁

## 线程池

### ThreadPoolExecutor构造函数 (高)

```
1 ThreadPoolExecutor(int corePoolSize,  
2                     int maximumPoolSize,  
3                     long keepAliveTime,  
4                     TimeUnit unit,  
5                     BlockingQueue<Runnable>  
workQueue,  
6                     ThreadFactory threadFactory,  
7                     RejectedExecutionHandler  
handler)
```

- `corePoolSize` 指定线程池的核心线程数(必须大于0), 核心线程就是一直在线程池里的长久存活的线程
- `maximumPoolSize` 指定线程池中的最大线程数(最大线程数>核心线程数), 临时线程用完销毁

- `keepAliveTime` 指定临时线程空闲时的存活时间
- `unit` 指定临时线程存活时间的单位
- `workQueue` 指定任务队列 (不能为null). 当提交的任务数超过核心线程数后, 再提交的任务就存放在工作队列
- `threadFactory` 指定哪个线程工厂创建线程 (不能为null)
- `handler` 指定线程忙, 任务队列满的时候, 新任务来了怎么办 (不能为null)

临时线程什么时候创建啊?

- 新任务提交时发现核心线程都在忙, 任务队列也满了, 并且还可以创建临时线程, 此时才会创建临时线程。
- 这样是最大限度避免创建线程

什么时候会开始拒绝任务?

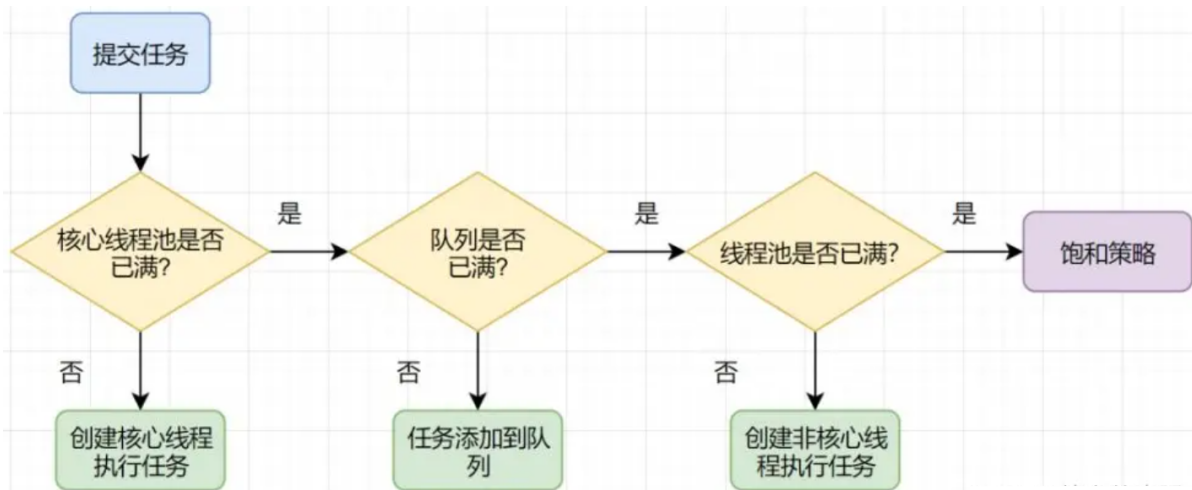
- 核心线程和临时线程都在忙, 任务队列也满了, 新的任务过来的时候才会开始任务拒绝。

## 提交一个新任务到线程池时, 具体的执行流程 / 线程池的工作流程 (高)

- 当我们提交任务, 线程池会根据`corePoolSize`大小创建若干任务数量线程执行任务
- 当任务的数量超过`corePoolSize`数量, 后续的任务将会进入阻塞队列阻塞排队
- 当阻塞队列也满了之后, 那么将会继续创建(`maximumPoolSize-corePoolSize`)个数量的线程来 执行任务, 如果任务处理完成, `maximumPoolSize-corePoolSize`额外创建的线程等待`keepAliveTime`之后被自动销毁



- 如果达到maximumPoolSize，阻塞队列还是满的状态，那么将根据不同的拒绝策略对应处理



## 线程池线程数如何设置? (高)

- CPU密集型: 任务需要大量计算, 很少阻塞, CPU一直处于忙碌状态. CPU核数 + 1
- IO密集型: 任务需要频繁的IO操作(与磁盘, 网络交互), CPU经常等待IO完成. CPU核数 \* 2

`Runtime.getRuntime().availableProcessors();` 获取CPU最大核心数

## 线程池中submit() 和execute()方法有什么区别 (中)

- 两个方法都可以向线程池提交任务
- execute()只能提交runnable任务, submit()既可以提交runnable任务, 又能提交callable任务
- submit()可以返回持有计算结果的Future对象.

## Executors中的常用线程池（高）

`static ExecutorService newCachedThreadPool()` 线程数量随着任务增加而增加，如果线程任务执行完毕且空闲了一段时间则会被回收掉。(全是临时线程, 没有核心线程)

`static ExecutorService newFixedThreadPool(int nThreads)` 创建固定线程数量的线程池，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程替代它。(全是核心线程, 没有临时线程)

`static ExecutorService newSingleThreadExecutor()` 创建只有一个线程的线程池对象，如果该线程出现异常而结束，那么线程池会补充一个新线程。(只有一个核心线程)

`static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)` 创建一个线程池，可以实现在给定的延迟后运行任务，或者定期执行任务。(使用 `ScheduledExecutorService` 类接受返回值, 调用 `schedule()` 方法来实现延迟执行, 调用 `scheduleAtFixedRate()` 方法或者 `scheduleWithFixedDelay()` 方法实现定期执行) 既有核心线程, 又有临时线程

注: Executors的底层其实也是基于线程池的实现类 `ThreadPoolExecutor` 创建线程池对象的。

## Executors创建线程池会存在什么问题（高）

Executors有 `newCachedThreadPool()`  
`newFixedThreadPool(int nThreads)`  
`newSingleThreadExecutor()` `newScheduledThreadPool(int corePoolSize)` 几大方法创建线程池。

Executors用起来比较方便,但是在大型分布式系统中直接使用会产生一些问题.

`newFixedThreadPool()` 和 `newSingleThreadExecutor()` 固定线程数量的线程池和单个线程的线程池,线程数量不会溢出,但是任务队列有可能溢出.允许的任务队列长度为`Integer.MAX_VALUE`,可能堆积大量任务,导致OOM.

`newCachedThreadPool()` 和 `newScheduledThreadPool()` 线程数量随着任务增加而增加,线程数量有可能溢出.线程数最大为`Integer.MAX_VALUE`,可能会创建大量线程,从而导致oom.

阿里巴巴开发手册中强制规定,线程池不允许使用Executors创建,而是通过ThreadPoolExecutor方式创建,这样可以使编程者更加明确线程池运行规则,避免资源耗尽.

## 线程池的拒绝策略 (中)

线程池的拒绝策略主要有四种:

1. AbortPolicy: 抛出`RejectedExecutionException`异常。
2. CallerRunsPolicy: 调用者线程自己执行任务。
3. DiscardPolicy: 直接丢弃任务,不处理。
4. DiscardOldestPolicy: 丢弃队列中最旧的任务,然后重新尝试提交当前任务。
5. 自定义拒绝策略,通过实现`RejectedExecutionHandler`接口,并重写`rejectedExecution`方法来实现自定义逻辑

## 同步 异步 阻塞 非阻塞 (中)

### 同步与异步

- **同步**：同步就是发起一个调用后，被调用者未处理完请求之前，调用不返回。
- **异步**：异步就是发起一个调用后，立刻得到被调用者的回应表示已接收到请求，但是被调用者并没有返回结果，此时我们可以处理其他的请求，被调用者通常依靠事件，回调等机制来通知调用者其返回结果。

同步和异步的区别最大在于异步的话调用者不需要等待处理结果，被调用者会通过回调等机制来通知调用者其返回结果。

### 阻塞和非阻塞

- **阻塞**：阻塞就是发起一个请求，调用者一直等待请求结果返回，也就是当前线程会被挂起，无法从事其他任务，只有当条件就绪才能继续。
- **非阻塞**：非阻塞就是发起一个请求，调用者不用一直等着结果返回，可以先去干其他事情。

**那么同步阻塞、同步非阻塞和异步非阻塞又代表什么意思呢？**

举个生活中简单的例子，你妈妈让你烧水，小时候你比较笨啊，在哪里傻等着水开（**同步阻塞**）

等你稍微再长大一点，你知道每次烧水的空隙可以去干点其他事，然后只需要时不时来看看水开了没有（**同步非阻塞**）

后来，你们家用上了水开了会发出声音的壶，这样你就只需要听到响声后就知道水开了，在这期间你可以随便干自己的事情，你需要去倒水了（**异步非阻塞**）

## 各种IO

(这里的各种IO比较复杂, 建议配合网上文章博客理解)

### BIO (Blocking IO)

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

1. **同步阻塞**：BIO是同步阻塞的，意味着每当有一个IO操作发生时，线程都会被阻塞，直到操作完成。
2. **面向流**：BIO面向流进行数据读写，数据处理通常在单个线程中完成。
3. **性能问题**：对于大量的并发请求，BIO会因为每个请求都需要独立的线程而性能受限。

客户端有连接请求时服务器端就需要启动一个线程进行处理

如果这个连接不做任何事情会造成不必要的线程开销，并且线程在进行IO操作期间是被阻塞的，无法进行其他任务

在高并发环境下，BIO的性能较差，因为它需要为每个连接创建一个线程，而且线程切换开销较大，不过可以通过线程池机制改善

## NIO (Non-blocking IO)

- 服务器端当accept一个请求后, 加入fds集合(文件描述符集合), 每次轮询一遍fds集合recv(非阻塞) 数据, 没有数据则立即返回错误, 每次轮询所有fd(包括没有发生读写事件的fd)
- 一个线程(在用户代码中)不断轮询检查各个socket是否有数据读入, 不断进行系统调用询问内核数据是否能够读写, 不断从用户态切换到内核态, 会很浪费cpu资源.
- 实现了一个线程处理多个IO操作, 但是性能依旧不好

## IO多路复用

IO多路复用是一种在单个线程中管理多个IO操作的技术。它允许一个进程或线程监视多个文件描述符, 并且在其中任何一个文件描述符就绪(可读、可写或异常)时执行相应的操作, 而无需阻塞其他操作

- 服务器端采用单线程通过select/epoll等系统调用获取fd列表, 遍历有事件的fd进行accept/recv/send, 使其能支持更多的并发连接请求
- 相比于BIO、NIO, IO多路复用最大的区别就是获取有事件的文件描述符方式发生了转变, BIO, NIO都是主动去轮询获取fd, 多路复用是将fd都交给内核去监控, 当某个fd能够读写时由内核告知应用程序处理, 这样就从主动轮询变成了被动通知了, 通过这种方式现在就可以一次处理多路IO了.

IO多路复用主要的功能就是实现通过内核对多个IO源的高效监听, 当某个文件能够读写时, 内核通知应用程序进行处理

## IO多路复用和NIO的区别

- NIO需要不停地循环检查各个socket是否有数据读入
- IO多路复用代码层面则不需要循环语句，虽然IO多路复用也是轮询，但是IO多路复用是交给内核进行各个socket的监控的

## 操作系统实现IO多路复用的技术

### select

- 是最早出现的 IO 多路复用技术，几乎所有的操作系统都支持。应用程序通过调用 select 函数，将需要监控的文件描述符集合（读集合、写集合和异常集合）传递给内核。内核会对这些文件描述符进行轮询，检查是否有文件描述符就绪

### poll

- 与 select 类似，也是通过内核轮询的方式来检查文件描述符的就绪状态。不同的是，poll 没有对文件描述符数量的硬限制，它通过一个链表来存储需要监控的文件描述符，理论上可以监控的文件描述符数量只受限于系统资源

### epoll

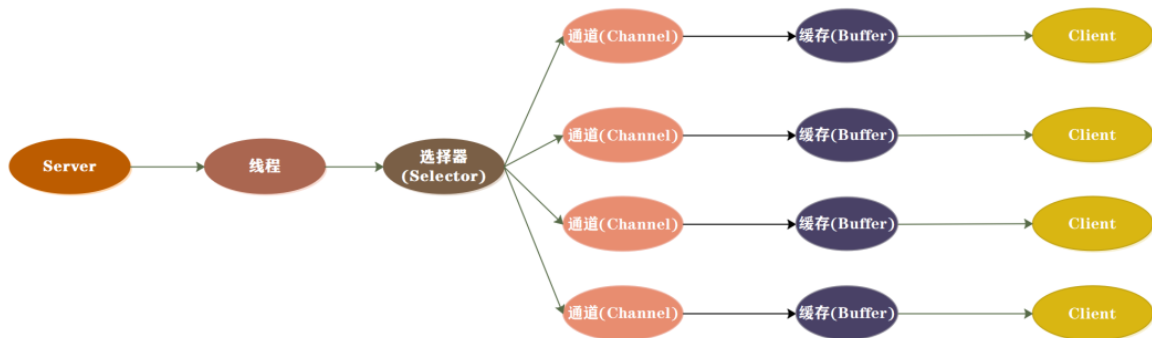
- 是 Linux 内核特有的 IO 多路复用机制，在处理高并发场景下表现出色
- epoll 采用事件驱动的方式，当有文件描述符就绪时，内核会通过回调函数将该文件描述符加入到一个就绪队列中。应用程序只需要从这个就绪队列中获取就绪的文件描述符，而不需要像 select 和 poll 那样进行轮询

**注意, 其实java的nio, 不是unix网络中定义的nio, 而是io多路复用. 同理netty也是基于io多路复用.**

线程可以对一些IO事件进行监听，并继续执行其他任务，不需要阻塞等待

当IO事件就绪时，线程会得到通知，然后可以进行相应的操作，实现了非阻塞式的高伸缩性网络通信

在NIO模型中，数据总是从Channel读入Buffer，或者从Buffer写入Channel，这种模式提高了IO效率，并且可以充分利用系统资源。



java的nio(io多路复用)主要由三部分组成 (了解即可, 不用记)

- 1. 通道 (Channel) :** Channel是**NIO**中用于数据读写的双向通道，可以从通道中读取数据，也可以将数据写入通道。与传统的IO不同，Channel是双向的，可以同时进行读写操作，而传统的IO只能通过**InputStream**或**OutputStream**进行单向读写。Java NIO中常见的Channel有：**FileChannel**（文件读写）、**DatagramChannel**（UDP协议）、**SocketChannel**（TCP协议）和**ServerSocketChannel**（监听TCP连接请求）等。
- 2. 缓冲区 (Buffer) :** Buffer是NIO中用于存储数据的缓冲区，可以理解为一个容器，可以从中读取数据，也可以将数据写入其中。Buffer具有一组指针来跟踪当前位置、限制和容量等属性。Java NIO中提供了多种类型的Buffer，例如**ByteBuffer**、**CharBuffer**、**ShortBuffer**、**IntBuffer**等。每种类型的Buffer都有自己特定的读写方法，可以使用 `get()` 和 `put()` 等方法来读写缓冲区中的数据。



3. **选择器 (Selector)** : Selector是NIO中用于监控多个Channel的选择器，可以实现单线程管理多个Channel。Selector可以检测多个Channel是否有事件发生，包括连接、接收、读取和写入等事件，并根据不同的事件类型进行相应处理。Selector可以有效地减少单线程管理多个Channel时的资源占用，提高程序的运行效率。