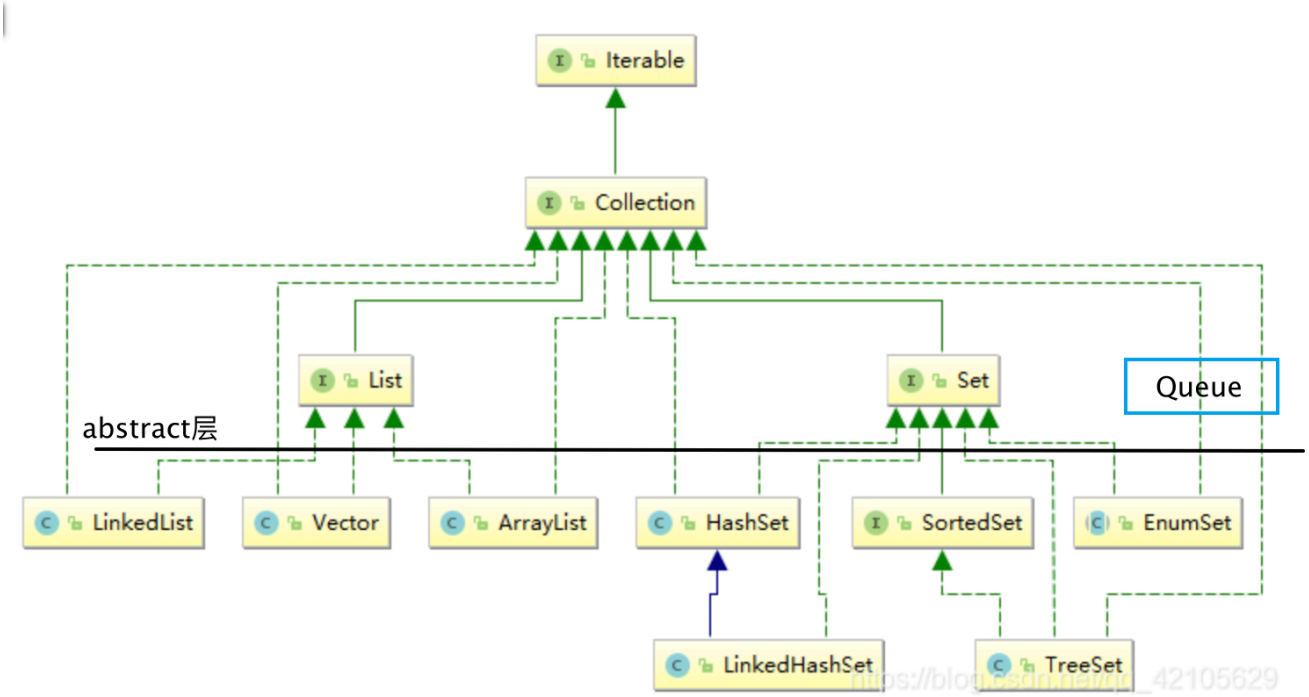
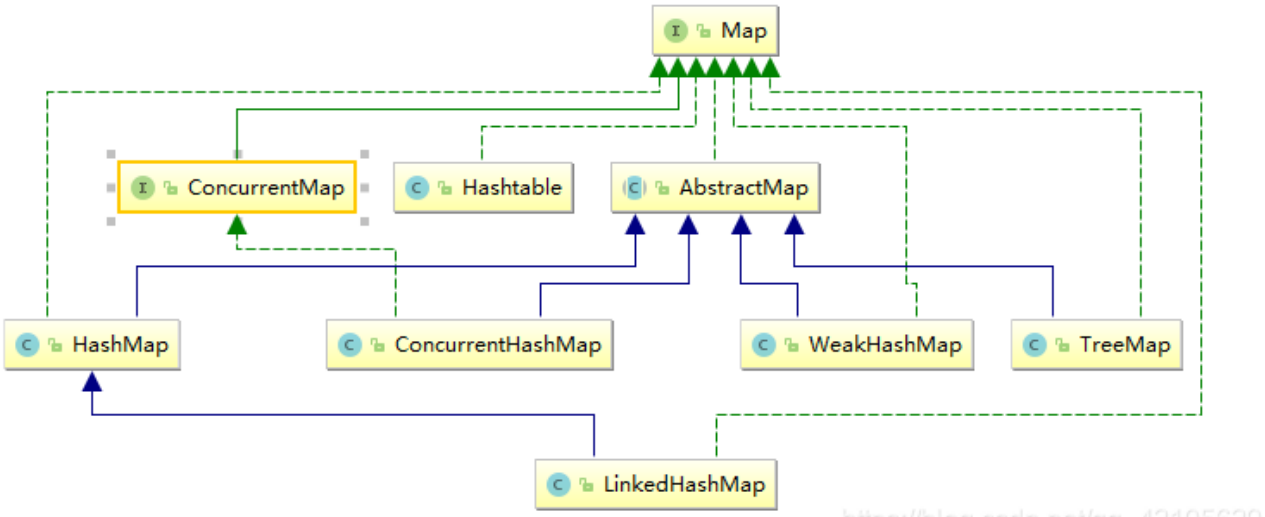
java数据结构

- 数据结构类图

> https://blog.csdn.net/qq_42105629/article/details/97545510





- Map

> Map是不是集合？https://blog.csdn.net/zimou5581/article/details/52706283/
>
> Map为什么可以用collections里面的方法？https://blog.csdn.net/sfhinsc/article/details/84060783

  - HashMap
    - 简单来说就是 HashMap底层使用数组来存储键值对, 当产生了Hash冲突的时候, 采用链表+红黑树来解决冲突，默认初始长度为16

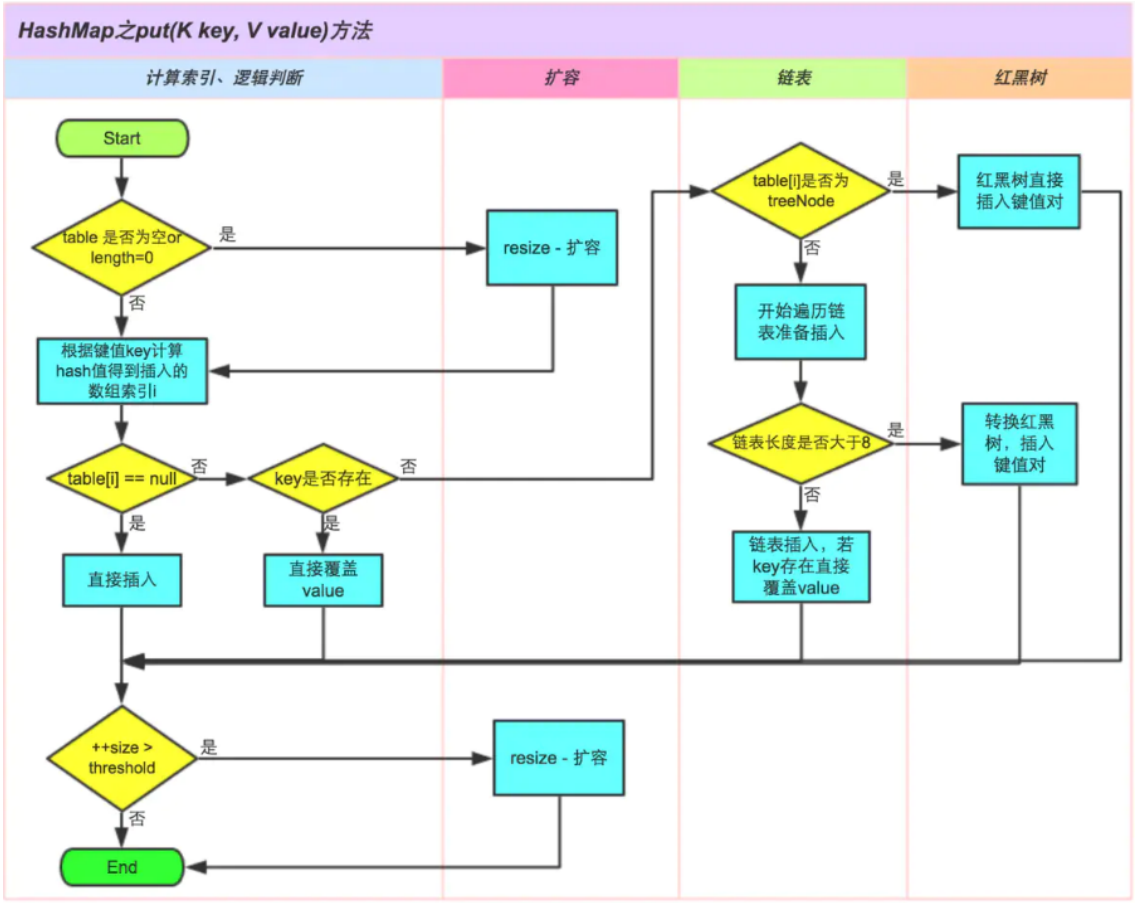
- 下面是HashMap内部重要方法的声明/源码

```
// jdk1.8
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    transient Node<K,V>[] table;

    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
        ....
    }

    static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V>
{
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;
        TreeNode(int hash, K key, V val, Node<K,V> next) {
            super(hash, key, val, next);
        }
        ...
    }

    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    }

    static final int hash(Object key) {
    }

    final Node<K,V>[] resize() {
    }
```

- putVal方法讲解

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //table为空或者length为0  ->  进行resize扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 根据key计算得到hash值，对应数组元素为空就直接插入
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
```

```
    else { //否则在链表或者红黑树中进行插入
        Node<K,V> e; K k;
        //判断当前节点的key是否和插入的key相同,如果是就直接进行替换
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        //判断是否是红黑树节点,如果是的话插入到树中
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
        else {//变量链表找到相等的节点
            for (int binCount = 0; ; ++binCount) {
                //遍历到链表尾仍没找到相同节点,就插入到链表尾
                //并判断是否达到了链表转红黑树的阈值,如果达到了进行转换
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
                        treeifyBin(tab, hash);
                    break;
                }
                //如果找到相同节点进行替换
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null
&&key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    //记录结构修改次数
    ++modCount;
    //map的键值对数量大于map...
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```



插入流程图[4]

- resize()方法详解

```
final Node<K,V>[] resize() {
    //首先保存原数组相关参数
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    //这里扩容有两种情况
    // i.原数组有值
    if (oldCap > 0) {
        //当原数组存有值的时候,分两种情况
        //a.原容量达到最大值,就不进行扩容了
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        //原容量没达到最大值,就扩容一倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    //下面两种都属于原数组未初始化,就根据默认数据和原数组参数给新数组参数进行赋值
    else if (oldThr > 0)
        newCap = oldThr;
```

```
    else {
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    //创建新的数组并指向它
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        //如果原数组不为空的话,遍历原数组元素并迁移到新数组里面
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                //首先标记原数组元素并释放原数组空间
                oldTab[j] = null;
                //根据元素的类型,选择迁移的方法
                //i.元素为单独节点,则直接rehash到新数组中
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                //i.元素为树节点,则进行红黑树的rehash操作
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                //iii.元素为链表,进行链表的rehash操作
                else {
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        //根据元素hash值&上原数组长度将原链表划分为2个链表,
                        //为1的部分一定是放在原索引+原数组长度的位置
                        //....解释下原理
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
```

```
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}
```

- hash方法实现 : hashcode+扰动操作
  - 扰动操作可以把高位和低位的特征结合起来, 降低哈希冲突的概率, 也就是说尽量让每一位的变化都可以对最终结果产生影响

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

    - 为什么要无符号右移16位?
      - 不做右移的话计算槽位的时候会**丢失高位特征**, 导致哈希碰撞概率增加
- 注意 : 提到了putVal() 的链表转红黑树时, 可以提一下对应的红黑树转链表
  - 这里可问 : 为什么两个阈值不同?
    - 设置为8是因为根据**泊松分布**, 为8的概率已经很小了, 所以在这种罕见的情况下再进行链表转红黑树
    - 设置为6是为了避免**频繁进行转换**
- 为什么用红黑树不用B+树?
  - 红黑树进行**修改**、**插入**、**删除**时，比B+树好**维护**很多
  - B+树**查询效率**略低于红黑树
  - B+树主要用于**数据存储在磁盘**的场景，其结构特点就是**矮胖**，每个节点可以存放一个磁盘大小的数据；而红黑树多用于**内存数据处理**
- 为什么不用平衡二叉树?
  - 红黑树基于平衡二叉树引入了**红黑节点**，在红黑树上进行插入删除的时候，不会像平衡二叉

树那样频繁调整，因为红黑树的**调整规则比平衡二叉树宽松**，可以说红黑树是在查询和插入删除的不断调整中做了一个**折中**

- 为什么负载因子（扩容因子）是0.75?
  - 低的话扩容频繁
  - 高的话冲突概率高
- $JDK1.8$ 和$JDK1.7$的对比
  -

主要区别	$JDK1.7$	$JDK1.8$
数据结构	数组+链表	数组+链表+红黑树
链表插入	头插法	尾插法
计算哈希	扰动操作更多	扰动操作更少

- HashMap线程不安全的原因?

> https://www.cnblogs.com/developer_chan/p/10450908.html

  - $JDK1.7$
    - 链表插入采用**头插法**, **翻转**了链表顺序, 导致多线程情况下可能出现死循环
  - $JDK1.8$
    - 多线程情况下会出现多个线程同时进行Put操作, 导致**数据覆盖**问题
- HashMap数组容量为什么一定是2的幂次方?
  - 这样可以提高HashMap计算key的哈希值的**速度**, 从而提高HashMap的存取速度
- 为什么String和Integer等包装类适合作为 key?
  - 因为这些包装类具有**不变性**, 从而保证key具有不变性
- 如何自定义类要作为key，需要重写哪些方法?
  - 需要重写 hashcode() 和 equals() 方法, 因为这两个方法在get() 和 put() 中有用到
- HashMap实现同步的方法
  - 使用 Collections.synchronizedMap(HashMap)
    - 该方法给传入的Map的所有操作都加上锁
  - 使用 HashTable
  - 使用 ConcurrentHashMap
- **总结** : HashMap底层是如何实现的?
  - HashMap底层数据结构是数组+链表, HashMap首先根据key的hashcode,然后通过扰动函数得到hash值, 再将hash值映射到数组中得到索引值, 如果当前索引无值就直接进行创建, 有值的话进行遍历, 找到相同的key后进行覆盖, 没找到的话进行插入; 1.8之后对扰动函数, 解决冲突的数据结构进行了优化, 也优化了链表插入的方式
- HashMap 和 HashTable 的区别?
  - HashMap 是线程不安全的, HashTable 是线程安全的, 因为HashTable 中每个方法都被synchronized修饰了
  - HashMap 效率比HashTable 高, 因为HashTable 被 synchronized修饰了, 并发效率降低

        - HashMap允许Key为NULL, HashTable,ConcurrentHashMap 不允许
            - 为什么这么设计?
                - HashMap是单线程, 可以通过ContainsKey()来判断Key是存在为NULL,还是根本就不存在, 然后再进行get或put操作; 但是HashTable和ConcurrentHashMap不行, 因为在使用了ContainsKey()之后, 有可能数据被其他线程修改了, 这个时候进行的get或put无法分辨是存在但为NULL, 还是说根本就不存在
            - 为什么只允许一个key为null?
                - 因为key要保证唯一
        - HashMap重新计算对象的Hash值, HashTable 直接使用对象的Hash值
- ConcurrentHashMap是如何实现的?

    > https://www.cnblogs.com/fsychen/p/9361858.html

    - 在HashMap基础上, 采用了CAS + synchronized来保证并发安全性
    - 具体来说, 首先所有的核心值采用**volatile**修饰, 保证多线程可见性, 在进行put操作的时候，在数组对应节点为空时, 不是直接插入, 而是采用CAS操作尝试写入, 失败就自旋; 不为空的时候, 采用synchronized锁写入数据, 其他操作同HashMap的put一样
    - ConcurrentHashMap 同 HashMap区别 : 从**并发**和**效率**方面来说
    - resize()

        > https://www.bilibili.com/video/BV17i4y1x71z?from=search&seid=9653973810102822900

        - ConcurrentHashMap的扩容实现可以对比hashmap的resize()实现来进行考虑，ConcurrentHashMap的扩容需要解决两个问题
            - 并发情况下，数组中每一个节点的迁移只能有一个线程来完成
            - 迁移的时候，如何控制其他线程的写入
        - 简单的解决思路
            - 让第一个调用扩容方法的线程来完成扩容操作，其他线程直接阻塞住，第一个线程操作的时候也是遍历数组的所有节点，对于每一个节点进行迁移，只不过在迁移之前，需要给每一个节点加锁，保证只有一个线程进行迁移
            - 当其他线程进行写入的时候，如果发现当前有值且正在被迁移或者迁移中的话，就阻塞住，否则就进行正常写入
        - 多线程优化
            - 由于其他调用扩容方法或者等待写入的线程处于阻塞状态，所以把其他线程加入到扩容中来，加快扩容的速度，这里实际上是把扩容中整个数组进行了划分，划分成了一段一段的，然后我们每个线程进入我们的transfer的时候，**会cas的获取属于自己的那部分迁移段**，然后进行迁移，迁移完成后判断是否迁移完成了，如果没有迁移完成，则会循环调用transfer方法，继续协助扩容
    - 计算size()
        - 如果用一个成员变量来统计元素个数的话，为了保证并发情况下共享变量的的安全，势必会需要通过加锁或者自旋来实现，如果**竞争比较激烈的情况下，size的设置上会出现比较大的冲突反而影响了性能**，所以在ConcurrentHashMap采用了**分片**的方法来记录大小
        - 每一次我们去put的时候，就会去尝试**cas的增加我们的basecount**，当我们**增加失败的时**

候，就把不断尝试**把增量cas加到CounterCell数组中的某个随机下标中**，当我们想要获取size的时候，就把basecount和CounterCell数组中所有元素做累计，然后返回

        - ~~sizectl~~
            - -1 初始化中
            - -n 正在扩容中，扩容线程数为n-1
            - 0 未被初始化
            - n 下一次扩容的阈值
    - LinkedHashMap
        - 在HashMap的基础上, 给每个节点加上了相当于before和after的指针, 并自带一个head, 从而进行维护
- List
    - ArrayList
        - 基于**数组**实现，默认初始长度为10
        - 关键点在于扩容 : 扩大为原来的 3/2
        - 特点
            - 查找快, 插入或删除慢
            - 线程不安全
            - 适合于顺序添加, 随机访问的场景
        - ArrayList与Vector的区别
            - Vector是线程安全的, ArrayList是线程非安全的 (Vector就是ArrayList的线程安全版本)
            - Vector可以指定增长因子, 否则默认为2
        - ArrayList在循环删除时会出现什么问题?
            - 采用for循环正向遍历删除时, 会**遗漏**连续的数据(因为后续元素会移动)
            - 采用for-each遍历删除时, 会报并发错误, 因为迭代器遍历需和迭代器删除搭配使用, 不能和List.remove搭配使用
        - ArrayList快速失败机制

            > https://blog.csdn.net/qq_41046325/article/details/89154887

            - 快速失败机制是java集合中的一种错误机制, 当**多个线程对同一个集合的内容进行操作**时, 就可能会产生fail-fast事件. ArrayList出现快速失败的前提是**使用了迭代器进行遍历**
            - ArrayList内部有一个**modcount记录表的修改次数**, **迭代器内部有一个expectmodcount记录迭代器期望的修改次数**, 当初始化迭代器的时候,expectmodcount会被赋值为modcount, 当每次**调用迭代器中的方法的时候, 就会检查modcount和expectmodcount是否相等**, 如果不相等的话就会抛出ConcurrentModificationException, 产生快速失败事件; 当我们调用ArrayList中的add或者remove的时候, 就会更改modcount, 此时再调用迭代器方法的时候, 就会被检查到不相等而抛出异常
    - LinkedList
        - 基于**链表**实现 (双向链表)
    - ArrayList和LinkedList区别

        > 从使用来说，如果没有插入删除场景，就用ArrayList，因为顺序存储，利于GC和CPU缓存的

        - 在大多数情况下, ArrayList查找快, LinkedList插入删除快
        - ArrayList消耗空间少, LinkedList消耗内存空间多

- HashSet
  - HashSet基于HashMap实现, 使用HashMap来存储对象, 即对象都放在了HashMap的key上, 而value采用一个static final 的object对象来标识
  - HashSet中的对象也需要注意正确重写其中的equals和hashCode, 以保证对象的唯一性
- LinkedBlockingQueue vs ArrayBlockingQueue

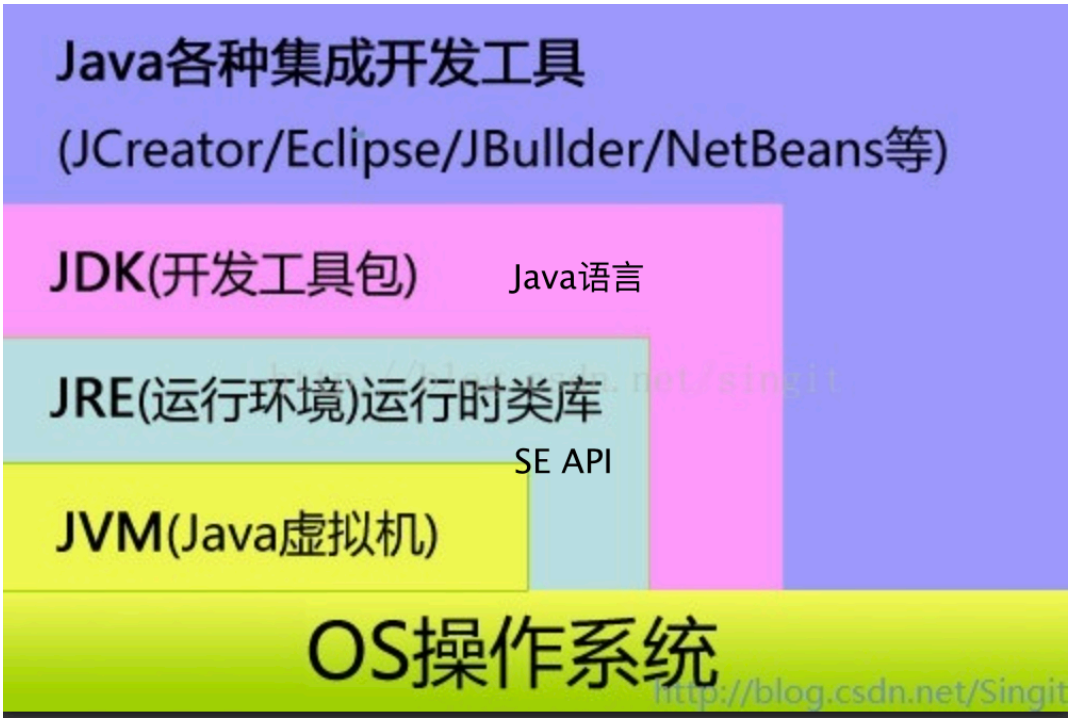
> https://www.zhihu.com/question/41941103
>
> https://stackoverflow.com/questions/11015571/arrayblockingqueue-uses-a-single-lock-for-insertion-and-removal-but-linkedblocki#

  - LinkedBlockingQueue：两把锁的ReentrantLock
  - ArrayBlockingQueue：一把锁的ArrayBlockingQueue，需要初始化时指定容量，环形存放
  - Array不需要创建Node节点，两把锁吃力不讨好

Java基础

- JDK & JRE

> https://blog.csdn.net/singit/article/details/62040688



- ~~展望Java未来趋势~~
  - 模块化，方便按需下载
  - 混合语言，多语言编程
  - 多核并行，同时利用多CPU执行任务
  - 丰富语法
- 编译型语言&解释型语言

> https://www.cnblogs.com/tsingke/p/12285650.html

  - 编译型语言：代码需要先经过**编译**转成平台相关的**机器语言**文件，运行时脱离开发环境，**效率比较高**
    - 比如C、C++

    - 解释型语言：使用**平台相关的解释器**将代码**解释为机器码**并执行，**效率比较低**，不过**可移植性强**
        - 比如Python、Java

        > Java既可以被认为是编译型，也可以被认为是解释型；因为它是先被编译成字节码，再被不同平台的JVM进行解释运行

- Java和C++的区别
    - 垃圾回收
    - Java泛型不允许使用基本数据类型
    - Java不支持操作符重载
    - Java是创建数组时分配存储空间, C++是定义数组时分配存储空间
- Java和Go的区别

    > https://www.nhooo.com/note/qa04vb.html

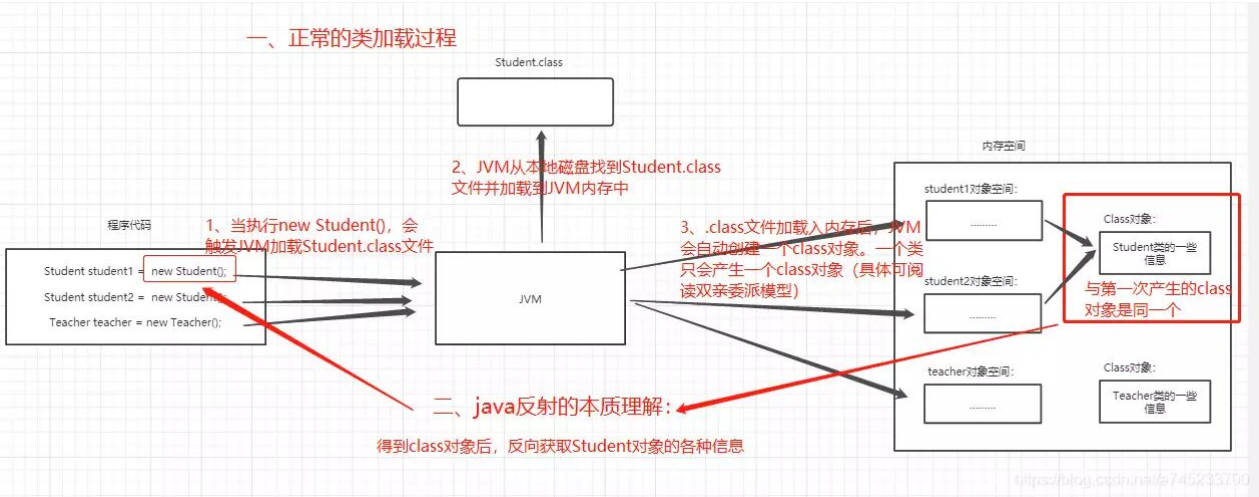
    - Java具有**异常**处理，而Go具有错误处理
    - Java支持**继承**，Go不支持继承
    - Java实现并发采用**多线程**，Go并发采用多协程和channel
    - Java的线程比较**重量**，一般最多上千个；Go的协程比较轻量，可以创建上百万个
- Java为什么能跨平台?
    - 因为Java把**编译和解释分开**来了, java源文件被编译器编译为字节码文件后, 通过不同操作系统上不同的JVM来进行解释运行字节码文件
- 面向对象是什么?

    > https://zhuanlan.zhihu.com/p/75265007

    - 面向过程：当解决一个问题的时候，**面向过程**会把事情**拆分**成： 一个个**函数和数据**，然后**按照一定的顺序**，**执行完这些方法**
    - 面向对象：当解决一个问题的时候，**面向对象**会把**事物抽象成对象**的概念，就是说这个问题里面有哪些对象，然后**给对象赋一些属性和方法**，然后**让每个对象去执行自己的方法**，问题得到解决
- 面向对象解决了什么问题?
    - 面向对象解决了面向过程程序的**耦合**强以及**维护性**差的问题
- Java语言特点
    - **继承**
        - 继承就是把多个类中共同的数据和逻辑(方法)**抽取**出来**封装**到一个新的类(即父类中), 然后让其他类来继承这个类, 那么这些共同的部分就不需要在子类中再次定义, 比如BaseDao, BaseAction这样的, 继承提高了Java程序的复用性和拓展性, 继承也是Java多态的前提
    - **封装**
        - 封装是指隐藏类的内部属性, 提供get, set来进行访问, 这样可以提高Java程序的安全性
    - **多态**
        - 多态是指**父类对象引用子类对象**, 发送消息给对象的时候, 对象自己决定如何响应. 这样可以让我们更加灵活的调用Java对象
        - 实现原理 : 我理解的多态是通过分派调用来实现的, 分派分为静态分派和动态分派

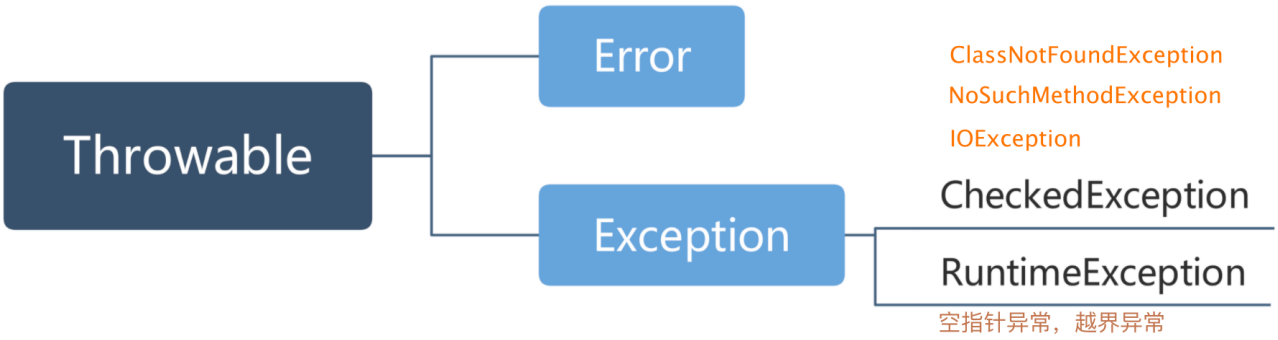
            > https://www.runoob.com/java/java-override-overload.html

    - 静态分派是依赖变量的**静态类型**(左边接收类型)来定位方法的执行版本, 比如**重载**, 其本质是**根据优先级进行方法匹配**（重载有的人认为不是多态、但其实我们可以认为重载是一个类的多态性体现）
    - 动态分派是依赖变量的**实际类型**(右边赋值类型)来定位方法的执行版本, 比如**重写**, 其本质是将类方法的**符号引用解析到了不同的直接引用**上
- 反射是什么以及对应的原理?
    - 反射是指在**程序运行期间**，**动态**获取**类信息**，**创建对象**，**执行方法**
    - 实现原理：在**类加载**时，会**注册class对象到内存**中，class对象中**包含了类的所有信息**，反射机制就是**获取class对象**，然后**拿到相关的信息**，并进行**动态创建、执行**



- Java异常

> https://blog.csdn.net/woshixuye/article/details/8230407



    - 受检异常/CheckedException
        - Java认为Checked异常都是**可以被处理的异常**，所以Java程序**必须显示处理**Checked异常；如果程序没有处理Checked异常，该程序在编译时就会发生错误**无法编译**
        - 对Checked异常处理方法有两种：
            - 当前方法知道如何处理该异常，则用**try...catch**块来处理该异常
            - 当前方法不知道如何处理，则在定义该方法是声明**抛出**该异常
    - 不受检异常/RuntimeException
        - Runtime如除数是0和数组下标越界等，其产生频繁，**处理麻烦**，若显示申明或者捕获将会对程序的**可读性**和**运行效率**影响很大。所以由系统自动检测并将它们交给**缺省的异常处理程序**；当然如果有处理要求也可以显示捕获它们
    - Error

        - 当程序发生**不可控的错误**时，通常做法是通知用户并中止程序的执行。与异常不同的是Error及其子类的对象不应被抛出。用于指示合理的应用程序**不应该试图捕获**的**严重**问题
- 异常时return返回值情况

> 基本数据类型，return复制值
>
> 引用类型，return复制地址（String被更改会改变指向，但是自定义类型不会）

```
public static T check(){
    T s = new T(); // 值A
    try{
        return s;
    }catch (Exception e){
        T.val = ... // 值B
    }finally {
        T.val = ... // 值C
    }
    return null;
}
```

    - T为基本数据类型，值为A
    - T为String，值为A
    - T为自定义类，值为C
    - 出现异常时，异常出现在try中return前时，返回null
- Java基本数据类型
    - byte : 1字节
    - short : 2字节
    - int : 4字节　　$-2^{31} ~~~2^{31}-1$
    - long : 8字节
    - char : 2字节
    - float : 4字节
    - double : 8字节
    - boolean : 单数据4字节, 数组里面数据1字节
- Java语法糖&原理
    - **泛型擦除**

        > https://www.cnblogs.com/wuqinglong/p/9456193.html

        - 什么是泛型擦除?
            - 泛型擦除是指编译器在编译时**擦除**了所有类型相关信息,所以在运行时不存在任何类型相关的信息
            - 帮助我们在限定语法层面传递的类型?
        - 泛型擦除存在的问题
            - 强制类型转换
                - 因为类型被擦除后, 所有的泛型类型变量最后都会被替换为原始类型, 因此在我们获取的时候就需要进行强制转换
                - 不过在存取变量的时候, 代码中已经帮我们自动转换了

        - 泛型类型变量**不能是基本数据类型**
            - 因为泛型类型变量被擦除后会替换为原始类型, 但基本数据类型没有对应原始类型,因此不能使用基本数据类型作为泛型类型变量
        - 引用传递问题
            - 不同类型进行引用传递是不被允许的, 父类引用传给子类的话自动转换会出现错误, 而如果是子类引用传递给父类的话违背了泛型设计初衷(解决类型转换问题), 因为这样我们还是需要自己进行类型转换
    - 解决思路

        ```
        <T extends Parent>
        ```

- 自动装箱、拆箱
    - 在编译时将其转化为对应的**包装**和**还原**方法
    - 比如：Integer.valueOf()、Integer.intValue()
- 遍历循环/ForEach
    - 将遍历代码转成**迭代器**的实现
- 条件编译
    - 根据布尔常量值的真假，编译器会把分支中不成立的代码块**消除**掉
- 变长参数
    - 实际是数组

- Object类下面的方法
    - hashCode()
        - 通过哈希算法返回对象的地址哈希值
    - equals()
        - 比较两个对象的地址是否相等, 一般比较地址是否相等意义不大, 需重写equals方法
    - toString()
    - getClass()
    - notify()
        - 唤醒在此对象监视器上等待的单个线程
    - notifyAll()
        - 唤醒在此对象监视器上等待的所有线程
    - wait()
        - 导致当前线程等待
- 为什么wait, notify方法在object类中，sleep方法在Thread类中?

    > https://blog.csdn.net/troubleshooter/article/details/99302283

    - wait, notify方法虽然是与线程相关的方法, 但他们不仅仅是普通方法, 更是**线程间的通信机制**; 比如现在有一个线程占有了A对象的锁, 当它释放锁的时候, 需要**通知其他等待该锁的线程**, 像这种场景, 就需要线程间的通信机制来完成, 而想要让所有对象都具备这种通信机制, 最好的方法就是**把wait, notify写在object类**中
- hashCode() 和 equals() 之间有什么联系?

    - 两个对象的 hashCode() 相同，equals() 不一定相同
    - 两个对象的 equals() 相同，hashCode() 一定相同
    - 重写对象的 equals() 时, 需要重写 hashCode()
        - 为什么? JAVA约定, 不这样的话对象放入HashMap会出现问题
- 原生hashCode实现
    - 源码实现

      > https://www.jianshu.com/p/be943b4958f4

        - 第一次调用的时候，取**线程随机值+3个固定值**组合成hashcode
        - 后续调用的时候，由于hashcode已经**存储在了hash中**，所以可以直接返回

> 注意：大部分基本类型、引用类型都重写了hashCode；
>
> 如果不重写，给你个随机值，也是很合理的

- ~~String的equals方法是如何重写的?~~

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

    - 简单来说, 先判断是否**地址是否相同**, 再判断**类型是否相同**, 类型相同就判断**string值是否相同**, 否则就返回不同
- String为什么是不可变的?
    - String的大多数属性都被**final修饰**, **私有**的且**没有提供修改方法** (虽然String不可变, 但其**指向的数组是可变的**, 私有的属性也可以通过反射进行获取)
    - 不可变的好处
        - 从**效率**(可直接缓存hashcode)和**安全**(防止被修改, 保证了HashSet的唯一性, 不可被写保证线程安

全)方面去解释

- String，StringBuffer，StringBuilder的区别
  - String不可变, StringBuffer**可变**
  - String修改会新建对象, StringBuffer修改**不会新建对象**
  - StringBuilder是StringBuffer的线程不安全版本
- 抽象类和接口的区别
  - 语法上
    - 继承抽象类的**关键字**是extends, 实现接口的关键字是implements
    - 一个类可以**实现多个**接口, 但只能**继承一个**抽象类
    - ~~抽象类中的**成员变量**可以是多种类型, 但接口中的成员变量必须是public static final~~
    - 抽象类中可以有方法体, 接口中不能有方法体, 但在JDK1.8之后, 接口可以定义default方法体
  - 语义上
    - 接口强调**应具备的功能**, 抽象类强调**所属关系**
    - 接口被用于常用功能, 便于日后**维护和修改**, 抽象类更倾向于充当公共类的角色, 便于代码**重用**
  - 举例：以AQS和Lock为例，大部分**子类实现了Lock接口**，而对于其中**lock方法的实现**，则是利用**内部类继承AQS并重写部分方法**完成；这个过程中，**AQS起到代码重用**的作用，**接口起到约束实现类功能**的作用
- 序列化

> https://blog.csdn.net/nobody_1/article/details/93506471

  - 对象的序列化是指**通过某种方式**把对象以**字节序列**的形式保存起来; 反之通过字节序列得到原对象就是反序列化
  - 序列化的对象需要实现**Serializable接口**
    - 该接口只是一个标识接口, 标识对象**可以进行序列化**
  - serialVersionUID是为了保证反序列化时找到正确的版本
  - 谁来完成序列化操作?
    - **对象流** / 自定义
- 创建对象的方法
  - new
  - clone
  - 反序列化
  - 反射
    - 反射机制原理：...
    - 反射通过调用类中的构造方法来动态创建对象
- static关键字

> https://blog.csdn.net/qq_34337272/article/details/82766943

  - 修饰**成员变量**或**成员方法** (静态变量/静态方法)
    - 被static修饰的成员, 不单属于这个类的某个对象, 而是**被类的所有对象共享**, ~~存储在**方法区中**.~~ 建议通过类名调用
    - 静态方法能被重写吗?

        - 所谓重写是针对对象来说的，对于类来说，没有限制子类不能和父类有同样的方法，具体采用哪个方法，由静态类型决定

```
class A {
    static void hello() {
        System.out.println("hello");
    }
}
class B extends A {
    static void hello() {
        System.out.println("hello2");
    }
}
A a = new B();a.hello(); // 输出hello
B b = new B();b.hello(); // 输出hello2
```

    - 修饰**代码块** (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**, **JVM在加载类的时候会执行静态代码块**, 该类无论创建多少对象, **静态代码块都只执行一次**
    - 静态~~**内部类** (用得少)~~
        - ~~用static修饰的内部类，称为**静态内部类，完全属于外部类本身**，不属于外部类某一个对象~~
    - 静态**导包**
        - 使用import static可以**指定导入某个类中的静态资源**, 使用的时候**不需要使用类名调用**, 可以直接调用
- final关键字

> https://www.cnblogs.com/swisszhang/p/9892309.html
>
> final修饰的东西，可能系统会有一定的优化?

    - 被final修饰的**类不可以被继承**
    - 被final修饰的**方法不可以被重写**
    - 被final修饰的**变量不可以被改变**
        - **不可变的是变量的引用**, 而**不是引用指向的内容**
        - final变量可以在**构造函数**里面**初始化/赋值**
- Java创建不可变类

> https://www.cnblogs.com/dolphin0520/p/10693891.html

    - **类**使用**final**修饰
    - **成员变量**使用**private** final修饰
    - **不设置setter**方法
    - 构造函数中的引用对象进行**深拷贝**
    - get方法不能直接**返回对象引用**
- native方法详解

> https://www.cnblogs.com/szlbm/p/5504603.html

> JNI : Java Native Interface

  - Java类 **声明 native**方法hello，以及**指定load的路径**
  - **JNI命令编译**该Java类，生成**.h文件**
  - 将.h文件**复制**到hello到c++实现项目里面
  - 将java安装目录下的 jni.h、jni_md.h 文件**复制**到该c++项目里（连接文件）
  - **生成ddl**文件，放到Java类**指定的load路径**，然后就可以**运行**该类了
- ~~数组初始化方式~~

> https://blog.csdn.net/weixin_38920324/article/details/79017075

```
int[] arrayA = new int[5];
int[] arrayA = {1,2,3,4,5};
```

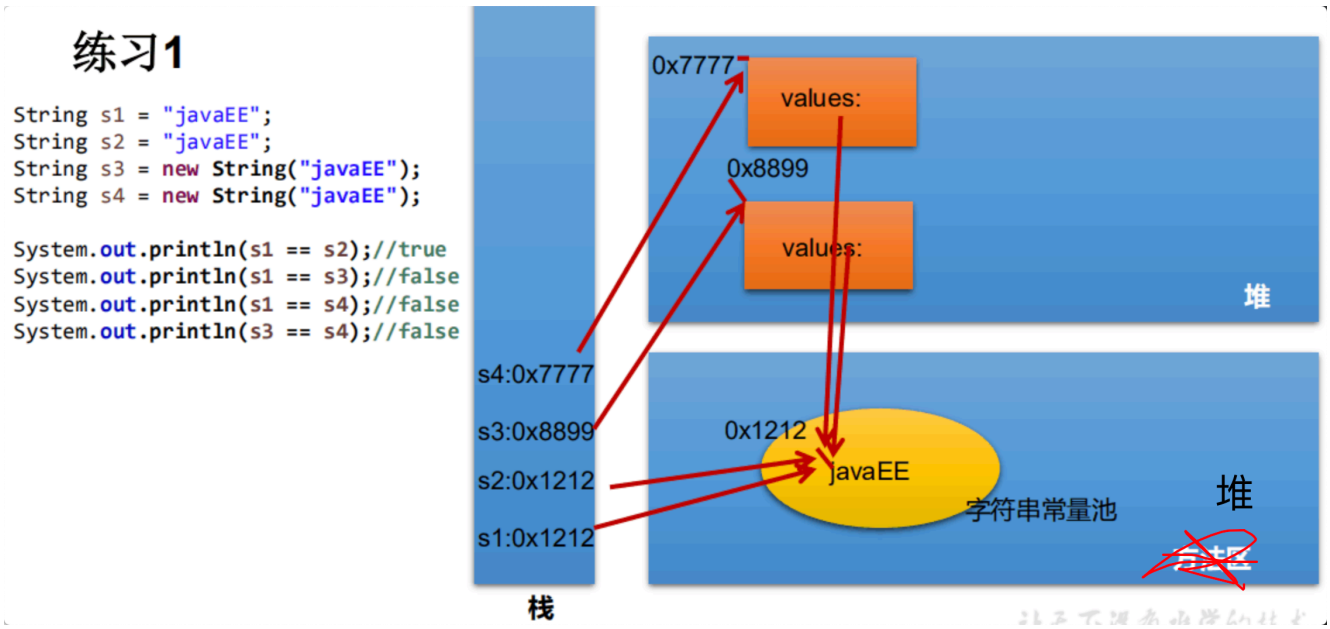
- ~~以下代码输出什么?~~

```
Map<Short, String> map = new HashMap<>();
  for(short i = 0; i <100; i++) {
     map.put(i, String.valueOf(i));
       map.remove(i-1);
  }
  System.out.println(map.size());
```

> 100; 因为类型发生了变化

- String & 常量池

> https://www.zhihu.com/question/36908414



  - 1.7后常量池到了堆里面去
  - new的时候，常量池没有不会在常量池里面新建，而是在堆里面新建
- Arrays.asList

> https://blog.csdn.net/kzadmxz/article/details/80394351

    - 得到的arrayList**长度是不可变**(利用了final来修饰)的，适用于**只读**的情况
    - 返回的arrayList是arrays里面的内部类，而非我们常用的arrayList
    - 想要变成可变可以这样做:

```
List<String> res = new ArrayList<>(Arrays.asList(items));
```

- 如何控制CPU占有率

> https://blog.csdn.net/smart_ferry/article/details/85345270

    - 单核如何控制50%?

```
while((System.currentTimeMillis()-startTime)<=50) ;
Thread.sleep(50);
```

> 空跑50ms，再休息50ms即可

    - 多核只需要开对应线程数来跑上面的代码就行了
- 浅拷贝和深拷贝

> https://www.cnblogs.com/shakinghead/p/7651502.html
>
> https://www.baeldung.com/java-deep-copy

    - 浅拷贝
        - 基本类型进行**值拷贝**，引用类型进行**引用拷贝**，拷贝**引用值/内存地址**
        - 实现方式
            - **标准构造**函数
            - 重写clone方法，调用super.clone()
    - 深拷贝
        - 基本类型进行**值拷贝**，引用类型进行**全量的对象拷贝**，拷贝**所有对象**
        - 实现方式
            - 更改构造函数，构造函数中涉及到对象的部分，**调用子类对象的构造函数**重新生成 新的对象
            - 重写clone方法，调用super.clone()后，还需要**调用成员变量中引用对象的构造函数**
            - 序列化&反序列化；**序列化到对象流**，然后**读取**出来
            - 反射；反射利用**无参构造**创建对象，然后**反射获取filed列表**，遍历每一个，对于基本类型就直接**get&set**，对于引用类型，**递归**获取数据，然后set
- Java注解
    - 注解作用：**为当前读取该注解的程序提供判断依据及少量附加信息**
- 三角关系：定义注解、使用注解、读取注解
- 注解实现原理：通过**反射**动态获取某个类、方法、变量上的注解，然后判断是否符合条件，并**进行相关处理**；一般**搭配AOP使用**
- 常见设计模式
    - 单例模式

> https://blog.csdn.net/fd2025/article/details/79711198

- 工厂模式（不太会...）

  > https://www.cnblogs.com/yssjun/p/11102162.html

  - 简单工厂：枚举
  - 工厂方法
  - 抽象工厂
- 观察者模式
  - **观察者模式**是一对多的模式，**一个被观察者**Observable和**多个观察者**Observer，**被观察者中存储了所有的观察者对象**，当被观察者接收到一个外界的消息，就会遍历广播**推送消息**给所有的观察者（主动式）
- 监听者模式
- spring中设计模式

  > https://zhuanlan.zhihu.com/p/66891710

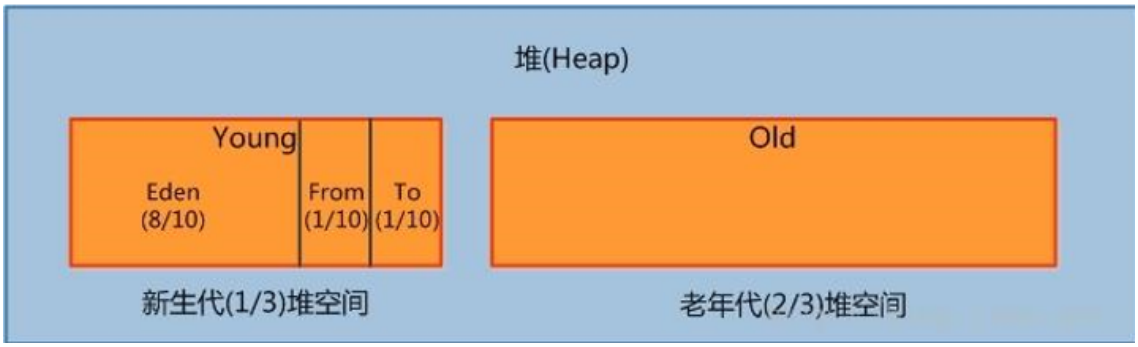
- 一般设置堆的大小
  - 最佳实践值：6GB
  - 过大：Full GC 耗时会很长
  - 过小：频繁发生GC

JVM

Java内存管理

- 运行时数据区域（JVM内存模型）

  - 程序计数器
    - 指示当前线程执行的**字节码行号**, 实际存储的是虚拟机字节码指令的**地址**

        - 两个作用
            - 字节码解释器通过改变程序计数器来**依次读取**指令, 从而实现代码的流程控制, 如顺序执行, 循环, 选择等
            - 在多线程的情况下, 程序计数器用于**记录当前线程执行的位置,** 从而在线程被切换回来的时候, 知道该线程上次运行到哪了
    - 虚拟机栈
        - 虚拟机栈描述的是**java方法执行的内存模型**, 每个方法在执行的同时会创建一个**栈帧**, 用于存储**局部变量表**, **操作数栈**, **动态链接**, **方法出口**等信息, 每个方法从调用到执行完成的过程, 就对应着一个栈帧在虚拟机中**入栈到出栈**的过程.
        - 虚拟机栈中的局部变量表还存放着编译器可知的基本数据类型(...), 以及对象引用类型, returnAddress类型
        - 栈过深会抛出StackOverflowError, 无法扩展时抛出OutOfMemoryError, 下同
    - 本地方法栈
        - 本地方法栈发挥的作用和虚拟机栈相似, 只不过虚拟机栈为虚拟机执行java方法服务, 本地方法栈为虚拟机执行**native方法**服务
    - 堆
        - 存放对象实例
        - 无法拓展抛出OutOfMemoryError.
        - JAVA堆从GC角度还可以细分为 如下
        - 

        - 为什么新生代存在两个survivor 区?

            > https://www.jianshu.com/p/2caad185ee1f

            - 回答前提：新生代采用的是复制算法
            - 一般复制算法是二等分，但是一般新生代对象**朝令夕死**，**没必要使用一半**来存储存活的，假设我们以8:2来划分，就发现一个新问题，在进行复制算法交换的时候，以2作为对象分配空间的时候，**很快就满了**，不太合适；而如果一直把对象在8里面分配，2里面迁移的话，又会**导致内存碎片**，但是，假如引入2个survivor，就能完美解决这个问题
    - 方法区
        - 存储已被虚拟机加载的**类信息**, **常量**, **静态变量**, **即时编译器编译后的代码**等数据
        - 无法分配内存时抛出OutOfMemoryError
- 方法区的变迁

> https://blog.csdn.net/qq876551724/article/details/78845366
>
> https://www.cnblogs.com/paddix/p/5309550.html
>
> https://www.zhihu.com/question/39990490

> 方法区只是一个**规范**/接口，具体实现根据虚拟机不同而不同

  - **JDK1.7**之前 利用**永久代**来实现，和堆相互隔离
  - **JDK1.7**时 把放在永久代中管理的字符串常量池，静态变量等**移出**了永久代
  - **JDK1.8**及之后 取消了永久代，利用元空间(meta space)来代替，在**本地内存**中实现/存储

> 为什么要永久代->元空间变迁?

  - **合并**不同虚拟机的**差异**
  - 字符串存在永久代中，容易出现**性能问题**和**内存溢出**

- 对象的创建过程
  - 首先将去检查能否在常量池中定位到该类的符号引用, 并**确认该类是否已经加载, 解析, 初始化**过, 如果没有的话, 先进行相应的类加载过程
  - 给新生对象**分配内存**
    - 内存分配的两种方式
      - 如果java堆内存规整的话，采用**指针碰撞**
        - 用过的内存在一边，没用过的内存在另一边，中间有一个分界指针，只需要向着没用过的内存方向将指针移动对象内存大小位置即可
      - 如果java堆内存不规整的话，采用**空闲列表**
        - 虚拟机会维护一个列表，该列表记录哪些内存块是可用的，在分配的时候，找一块足够大的内存块分给对象实例，然后更新列表记录
      - 堆是否规整取决于使用的GC收集算法
    - 内存并发问题
      - 虚拟机在创建对象的时候，必须要保证线程安全，虚拟机创建对象时采用两种方法来保证线程安全
      - CAS+失败重试： 采用CAS操作保证创建对象的原子性
      - TLAB：为每一个线程预先在Eden区分配一块内存，JVM在给线程中对象分配内存的时候，优先在TLAB中进行分配，不够时再使用前面的CAS操作进行内存分配
  - 把分配的内存空间都**初始化为零值**
  - 设置对象的**对象头信息** : 如这个对象是哪个类的实例, 对象的GC年龄, 如何才能找到类的元数据信息, 对象的哈希码等
  - 执行**自定义的初始化**方法
- 对象内存布局
  - 对象头
    - Mark Word：GC年龄, 哈希码, 锁标志
    - 类型指针
  - 实例数据
    - 对象真正存储的数据
  - 对象填充
    - 占位符
- 对象访问方式/访存定位
  - 通过**直接指针**访问对象

        - 这种情况下局部变量表中的引用类型存储的就是**堆中实例对象的地址**, 只不过堆中的实例对象不止包含对象实例本身, 还包含对象类型~~在方法区中的地址~~
    - 通过**间接句柄**访问对象
        - 这种情况下Java堆中会划分出一块内存作为句柄池, 局部变量表中的引用类型存储的就是**对象的句柄地址**,而句柄中包含了对象实例数据和对象类型数据各自的地址信息
    - 两种方式的比较
        - 使用句柄的话, 在对象移动的时候, 只需要修改句柄中的指针即可, 不需要修改reference本身
        - 使用直接指针的话, 节省了一次指针定位的时间开销
        - HotSpot使用的是直接指针, 但很多框架和语言使用句柄访问
- OOM解决方案
    - 我们得知道, 除了程序计数器之外的其他运行时区域, 都有可能会发生OOM
    - 我们第一步应该分析OOM的错误信息, 从而得知异常出现的区域, 然后不同区域进行不同处理
        - Java堆溢出
            - `java.lang.OutOfMemoryError: Java heap space`
            - 出现堆溢出, 可以使用**内存映像分析工具**对堆转储快照进行分析, 重点是分清楚是出现了**内存泄漏**还是**内存溢出**
                - 内存泄漏（对象不再会被使用，但是可达）
                    - 典型场景：**IO流不close**，因为进程可用文件描述符有限，不close的话会导致后续依赖文件描述符的行为无法进行，进而发生资源泄漏，导致内存泄漏
                    - 可进一步使用工具查看GC Roots的**引用链**, 从而定位出泄漏代码
                - 内存溢出
                    - 代表内存中对象确实都还应该活着, 此时应检查内存是否可以调大, 是否存在某些对象生命周期过长, 持有状态时间过长的情况
        - 虚拟机栈和本地方法栈溢出
            - 在这块区域, 一般产生StackOverflowError, 通常可以**通过错误信息直接定位问题**所在.
            - 当建立过多线程的时候, 也是有可能会出现OOM的, 在不能减少线程数的情况下, 只能通过减少栈容量来解决内存溢出
        - 方法区溢出
            - `java.lang.OutOfMemoryError: PermGen space`
        - 根据错误信息检查，大概率是常量赋值错误，或者生成了过多的类
- 垃圾回收机制
    - 垃圾回收机制指对内存中已经死亡或者长时间没有使用的对象进行清除和回收, 从而释放空间, 防止内存泄漏
- 如何判断是否需要回收?
    - **引用计数法**
        - 当对象被其他对象引用的时候, 就对该对象的引用计数+1, JVM对引用计数为0的对象进行回收
        - 缺点 : 无法区分**循环引用**的对象
    - **可达性算法**
        - 通过**GC ROOT**为起始点, 找它引用的对象, 当一个对象不可达的时候, 它就是可回收对象
        - 注 : GC ROOT包括如下

        - 虚拟机栈中**本地变量表**引用的对象
        - 方法区中**类静态属性**引用的对象
        - 方法区中**常量**引用的对象
        - **本地方法栈**中JNI引用的对象
- 四种引用

> https://blog.csdn.net/jiangxiulilinux/article/details/105391516

    - 强引用（new）
        - GC永远无法回收
    - 软引用（缓存）
        - 可以逃脱GC回收直至JVM耗尽内存
    - 弱引用（ThreadLocal）
        - 下次GC就会回收
    - 虚引用（堆里面存储了直接内存的引用）
        - 虚引用和没有引用一样，任何时候都可能被垃圾回收；虚引用须和引用队列联合使用，当垃圾回收器准备回收对象前，如果发现对象还有虚引用，就会在回收前，把这个虚引用加入到相关的引用队列中。程序如果发现某个虚引用被加入到引用队列中，就可以在对象被回收之前采取行动
        - 虚引用必须和引用队列一起使用，它的作用在于**跟踪垃圾回收过程**
- 垃圾回收算法
    - **标记-清除**
        - 标记**需要回收**的对象, 标记完成后统一清除
        - 缺点
            - 效率低
            - 产生碎片
    - **复制**
        - 将内存按一定比例划分成两块, 每次使用其中一块, 当这一块的内存用完后, 就把还存活的对象**复制**到另一块内存中, 再把已使用的内存一次清理
        - 优点
            - 解决了内存碎片化的问题
            - 顺序分配内存, 简单高效
            - 适用于对象存活率低的场景
        - 缺点
            - 不适用于对象存活率高的场景
    - **标记-整理**
        - 标记需要回收的对象, 把存活对象向一端**移动**, 然后清理端边界外的对象
        - 优点
            - 避免了内存的不连续性
            - 适用于存活率高的场景
    - **分代收集**
        - 根据对象生命周期的不同将内存划分为不同的区域, 一般分为老年代和新生代, 新生代对象存活率低, 采用复制算法, 老年代对象存活率高, 采用标记-清除或标记-整理

- 为什么标记整理和标记清除都是标记可回收对象(垃圾)

  > https://blog.csdn.net/shenTiBeiTaoKongLa/article/details/106980813

  - 因为如果标记不可回收对象，那么在清除的时候，就会把非不可回收对象进行清理，但是非不可回收对象中可能存在一些还存活，但只是我们没有标记上的对象(因为新加入老年代的原因)
- 新生代为什么不使用标记整理

  > https://www.zhihu.com/question/264981915

  - 标记比较麻烦
  - 整理比较麻烦(考虑位置是否有值/活的对象移到一边，需要进行交换)
- 垃圾回收器的并发与并行
  - 并行
    - 指多条垃圾收集线程**并行**工作，但此时用户线程仍处于**等待**状态
  - 并发
    - 指用户线程与垃圾收集线程**同时执行**（但不一定是并行的，可能会交替执行），垃圾回收线程在执行时不会停顿用户程序的运行
- GC的分类
  - Minor GC
    - 当**Eden区空间不足以分配对象**时，触发Minor GC
    - 过程如下
      - 复制 : 把Eden, From Survivor中存活的对象复制到 To Survivor中, 并把这些对象的年龄+1, 如果达到了老年标准或者To Survivor放不下了就放到老年代
      - 清空 : 清空Eden, From Survivor中的对象
      - 互换 : From Survivor和To Survivor互换, 原To Survivor成为下一次的From Survivor
  - Full GC
    - 调用System.gc时, 系统建议执行Full GC
    - 老年代空间不足的时候
    - 老年代最大连续空间小于晋升到老年代对象的平均大小
    - Eden, From Survivor向To Survivor复制时, 对象大小大于To Survivor, 且大于老年代
- GC中Stop the world
  - 即在执行垃圾收集算法的时候, 除了GC线程之外, 其他线程都会被挂起, 等待GC线程执行完毕后才能再次运行 (多数GC优化通过减少stop the world的时间来提高程序性能)
- 各垃圾回收器的特点及区别
  - 年轻代
    - Serial 回收器
      - **单线程回收**, 进行回收的时候需要暂停其他工作线程
      - **简单高效** , **Client**模式下默认的年轻收集器
    - ParNew 回收器
      - **多线程**版的Serial
      - 适合运行在**Server**模式下
      - 可与CMS收集器**搭配**工作
    - Parallel Scavenge 回收器

        - **多线程**
        - 注重系统的**吞吐量**
        - 适合在后台运算而不需要太多交互的任务
    - 老年代
        - Serial Old 回收器 (标记-整理)
            - 特点与年轻代差不多
        - Parallel Old 回收器 (标记-整理)
            - 特点与年轻代差不多
        - CMS 回收器 (标记-清除)
            - **并发收集**
            - 注重GC**停顿**时间
            - 执行步骤
                - 初始标记 (STW)
                - 并发标记
                - 重新标记 (STW)
                - 并发清除
            - 缺点
                - 占用cpu
                - 产生浮动垃圾，需预留空间
                - 空间碎片多
    - G1
        - G1也是一个非常关注延迟的垃圾收集器, 它最有意思的地方就是把Java堆划分为多个大小相等的独立区域, 回收时以分区为单位进行回收
        - **并行和并发** : 使用多CPU来缩短STW的时间, 与用户线程并发执行
        - **分代收集**
        - **空间整合**
        - **可预测停顿时间**
            - 因为G1会跟踪各个Region里面的垃圾堆积的价值大小, 在后台维护一个优先队列, 每次根据允许的收集时间, 优先回收价值大的Region, 从而保证G1能在有限时间内获取尽量高的收集效率
        - 执行步骤
            - 初始标记
            - 并发标记 , 除了该阶段以外, 其余阶段都会产生STW
            - 最终标记
            - 筛选回收
- 给对象进行内存分配策略
    - 优先Eden区
    - 大对象进行老年代
    - 长期存活进入老年代
    - 动态年龄判断
- JDK1.8默认使用Parallel Scavenge&Parallel Old
- JVM调优

> 查看某个Java进程内的线程堆栈信息

class文件与类加载

- Class文件格式
  > 16进制案例： CA FE BA BE 00 00 00 32
  - 头四个字节：魔数（ Magic Number ），用于确定这个文件是否为一个能被虚拟机接受的Class文件
  - 接下来的四个字节：次版本号+主版本号（45开始）
- 类加载的过程
  - 加载
    - 通过类的**全限定名**获取类的二进制**字节流**
    - 把字节流代表的**静态存储结构**转化为方法区的**运行时数据结构**
    - 在内存(**堆**)中生成一个代表这个**类的class对象**, 作为方法区这个类的各种数据的**访问入口**
  - 连接
    - 验证
      - 通过各种**验证**确保class文件的字节流中包含的信息符合虚拟机要求, 不会危害虚拟机自身安全
    - 准备
      - 为类变量**分配内存**并**设置零值**
    - 解析
      - 虚拟机将常量池内的**符号引用**替换为**直接引用**的过程
  - 初始化
    - 使用**自定义的值**去初始化**类变量**和其他资源
    - 初始化时机（这里的初始化不是我们理解的对象初始化构造函数，而是**收集静态变量的值，组成的一个cinit类初始化，进行赋值**）
      - 遇到new, getstatic, putstatic等字节码指令时
      - 反射调用类时
      - 初始化类时发现父类没有初始化
      - 虚拟机启动时, 指定的主类(main类)
  - 使用
  - 卸载
- 加载类时，数据的加载顺序
  > https://blog.csdn.net/w893932747/article/details/86180500
  - 类加载时，**只会加载类级别的变量**
  - **静态变量和静态代码块是按序执行**
  - **父类先于子类被加载**
- 类的元数据和Class对象
  > https://www.zhihu.com/question/38496907
  > Class对象其实可以理解为**对方法区里面的元数据的一种封装**，通过Class对象，提供一种API来进行**类信息的访问**；虚拟机那本书上的对象访存方法应该是1.6的，因为1**.6之前Class对象在方法区**，1.6后就在堆了，而对象类型指针，指向的应该是Class对象

    - Java里面的Class在运行时是包含两部分数据的,一部分是类的**元数据**信息,这部分数据存储在**方法区,**包括常量池、属性表、方法表和异常表等,JVM在加载完一个类后会创建一个java.lang.Class类型的实例，这个实例是存储在堆中；除了基本数据类型外,java中所有的对象都是在堆中创建的。

      对于静态变量,它的引用是存储在方法区的,但是它指向的对象还是在堆中创建的，比如:

      `static int i = 1; //1这个值是存储在方法区static Object o = new SomeObject(); //`o这个引用是存储在方法区,但是SomeObject()这个对象却是存储在堆中的

- 类加载器
    - 启动类加载器
        - 加载Java核心类
    - 扩展类加载器
        - 加载JRE拓展目录
    - 应用程序类加载器
        - 加载用户类路径上指定的类
- 双亲委派模型
    - 子类加载器收到加载请求, 不会先去处理, 而是交给父类加载器处理, 父类加载器处理不了再交给子类加载器; 这样可以更加**安全**, **有层次**
- 破坏双亲委派模型的例子

  > JDBC和双亲委派模型
  >
  > https://blog.csdn.net/sinat_34976604/article/details/86723663

    - JDBC让**父类加载器调用子类加载器来加载类**, **破坏**了双亲委派模型, 避开了双亲委派模型的弊端
    - 在双亲委托模型下，类加载器是由下至上的，即下层的类加载器会**委托上层**进行加载。但是对于SPI来说，有些接口是**JAVA核心库提供**的，而JAVA核心库是由**启动类加载器**来加载的，而这些接口的实现却来自于不同的jar包（厂商提供），JAVA的启动类加载器是不会加载其他来源的jar包，这样传统的双亲委托模型就无法满足SPI的要求。而通过给当前线程设置上下文类加载器，就可以设置的上下文类加载器来实现对于接口实现类的加载
- ~~Tomcat类加载器机制~~

  > https://www.jianshu.com/p/51b2c50c58ebhttps://www.iteye.com/blog/w574240966-2152785

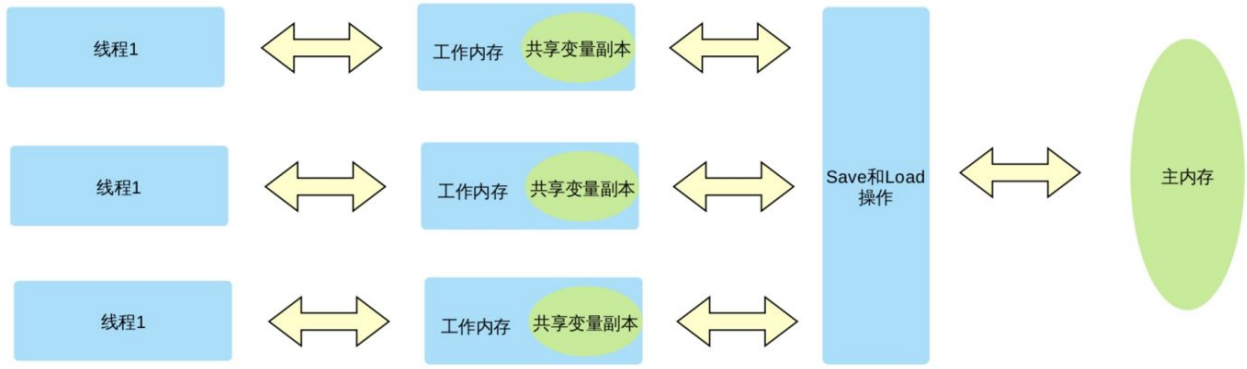
    -
    - tomcat 的类加载机制与java的类加载委派模型不同之处就是，当WebappN被请求加载一个类的时候，并不一定完全会按照 java类加载器的委托模式，WebappN首先会在本地库中查找而不是直接委托。但是也有例外.

并发

内存模型

- Java内存模型：定义程序中各个变量的**访问规则**

  > https://zhuanlan.zhihu.com/p/145902867



> 关于主内存和工作内存之间的交互协议，Java内存模型有定义**8种具体的原子操作**(比如read,load,use,store等)以及多种操作规则

- 并发编程三大特性

  > https://blog.csdn.net/emmmsuperdan/article/details/81564412

  - **原子性**
    - 操作要么执行，要么不执行
    - Java内存模型保证了**基本数据类型**的访问和读写是原子的
    - 但是**更大范围**的原子，需要**自己**并发编程保证
  - **可见性**
    - 修改能立即被其他线程**知晓/可见**
  - **有序性**
    - Java内存模型允许编译器/处理器对指令进行**重排序**，重排序符合**一定规则**(先行发生规则)，**不会影响单线程**，但是会影响多线程
- happen-before原则

  > https://baijiahao.baidu.com/s?id=1628346233476376109&wfr=spider&for=pc
  > 可以理解为对编译器优化代码的约束，让线程之间遵循这些规则

  - **程序次序**规则：**单线程**内保证**代码执行结果不变**/有序
  - **管程锁定**规则：多线程情况下，**某个线程解锁**后，**另一个获取锁的线程**能**看到之前线程**的操作
  - **线程启动/终止**规则：父线程执行中，启动子线程，则**子线程可见父线程启动子线程前的操作**；父线程执行中，子线程结束，则**父线程可见子线程结束前的操作**
  - **线程中断**规则：**对线程interrupt()方法的调用**，happens-before于**被中断线程的代码检测到中断事件的发生**（即interrupt后会刷新工作内存，然后子线程读取最新的）
  - **传递**规则：happens-before原则具有传递性，即A happens-before B ， B happens-before C，那么A happens-before C
- volatile

  > https://zhuanlan.zhihu.com/p/53415249
  >
  > https://www.infoq.cn/article/java-memory-model-2/

  > 对于volatile，Java内存模型定义了**特殊的访问规则**(基本的传递等规则没有列出来)：
  >
  > 1. **使用**(use)值前，必须从主内存**刷新**(load)最新的值

> 2. **修改**(assign)值后，必须立刻**同步**(store)回主内存中
> 3. **修改**(assign)volatile变量时，确保volatile**写之前**的操作不会被编译器重排序到volatile**写之后**（lock指令/内存屏障）
> 4. **读**(read)volatile变量时，确保volatile**读之后**的操作不会被编译器重排序到volatile**读之前**

- volatile的作用
  - 保证此变量对**所有**线程的**可见性**
  - **禁止指令重排序优化**

    > https://zhuanlan.zhihu.com/p/53415249
    >
    > 简单理解：单线程情况下存在的依赖性，在多线程情况下不存在了，多线程情况下重排序后，就会导致并发问题(并发编程-p29页)

- volatile使用场景
  - **CAS操作**
  - **延迟加载单例模式**
- volatile缺点
  - 不能保证复合操作的原子性，比如 i++

线程安全&锁

- 线程安全是什么?
  - 一个类或方法被**多个线程访问**并**正确运行**就叫做线程安全
- 如何实现线程安全?
  - **互斥同步**
    - 临界区：volatile
    - 互斥量：Synchronized / ReentrantLock
    - 信号量：Semaphore
  - **非阻塞同步**
    - CAS
  - **无同步**
    - 线程本地存储：ThreadLocal
- CAS & Atomic类
  - CAS是一种基于乐观锁的**比较交换**技术, CAS操作包含**内存地址**, **原值**, **新值**三个操作数, 如果内存地址里面的值和原值一样, 就进行更新, 否则进行**自旋**, 等待下一次执行
  - CAS操作的问题
    - **ABA问题**
      - JAVA提供了AtomicStampedReference类进行**版本记录**, 解决了该问题
    - **不能保证代码原子性**
      - CAS只能保证**一个变量的原子性**操作, **不能保证整个代码块的原子性**, 涉及到代码块的时候需要使用synchronized
    - **CPU使用率增加**
      - CAS是一个循环判断的过程, 没有成功会**一直占用CPU资源**

    - CAS实现原理
        - CAS操作会被编译成一条处理器指令, 比如**cmpxchg指令**
- synchronized

> https://blog.csdn.net/qq_36526036/article/details/104782986
>
> 作用：确保多个线程在同一时刻, 只有一个线程处于方法或同步块中, 它保证了线程对变量访问的**可见性**和**排他性**

    - synchronized的特征
        - **自动释放锁**
        - **可重入**
        - ~~不具备继承性, 子类重写父类带synchronized的方法时, 也需要加上synchronized~~
        - ~~synchronized(this) 锁定的是当前对象~~
    - 实现原理
        - 如果是**修饰对象**，那么在编译的时候，synchronized括号开始和结尾会被分别编译成**monitorenter**和**monitorexit**指令，执行monitorenter的时候会尝试去获取monitor对象，获取成功就相当于占用锁成功，获取失败则阻塞等待，当调用monitorexit的时候，相当于释放锁
        - 如果是**修饰方法**，那么就是在编译的时候添加一个**ACC_SYNCHRONIZED标识符**，运行到方法的时候就会调用相应的同步方法
        - 两种情况本质没有区别，一个隐式，一个显式
    - synchronized的用法
        - 修饰一个**代码块**
        - 修饰一个**非静态方法**
        - 修饰一个**静态方法**
    - synchronized的**不足**
        - 如果获取锁的线程由于等待IO或者调用sleep方法被**阻塞**了, 但是**又没有释放锁**, 其他线程**只能一直等着**, 因为synchronized在获取锁的时候**不能设置等待时间**
    - ~~synchronized一定线程安全吗?~~ （这里对于线程安全的定义没有做较好的阐释，所以是假问题）

> ~~https://blog.csdn.net/qq_41908272/article/details/90515342~~

        - ~~不一定, 当一个线程调用synchronized修饰的普通方法, 另一个线程调用synchronized修饰的静态方法时, 就会出现线程不安全的问题, 因为两个synchronized对应的Monitor对象是不一样的~~
    - volatile和synchronized区别
        - volatile**轻量级**, 只能修饰**变量**; synchronized**重量级**, 还可修饰**代码块或方法**
        - volatile不能保证**复合操作的原子性**(i++), synchronized可以保证
        - ~~synchronized使用更加广泛~~
    - synchronized和Lock(ReentrantLock)的区别
        - synchronized是java**关键字**, Lock是java**接口**
        - synchronized**无法判断是否获取到了锁**, Lock**可以判断是否获取到了锁**
        - synchronized会**自动释放锁**, Lock**不会自动释放**, 需在finally中手动释放
        - Lock锁可以**设置等待时间**, 到了时间自动放弃获取锁
        - Lock内有丰富的**关于锁的函数**
        - 如何选择？

        - ~~资源竞争不激烈的情形尽量使用synchronized~~ 正常情况下都=用synchronized
        - ~~竞争激烈的情形使用Lock~~ 涉及到Lock独有功能时，用Lock
- 锁升级的过程
    - 首先我们知道对象头里面存储着MarkWord(hashcode，GC年龄，锁标志)、class对象
    - 每一次获取锁的时候，首先就会把对象头里的数据(**不包括锁标志位**)存储到栈，CAS更新markword(具体更新的是什么需要根据锁类型来决定)，来代表获取了锁
    - 偏向锁获得&撤销
        - 线程CAS更新markword为**线程ID**，更新成功就设置为偏向锁，后续如果该线程获得锁的时候，**直接判断**是否开启了偏向锁，以及是否是当前的线程，然后直接进入方法；如果CAS**设置失败**，就发起锁撤销，**暂停线程**，如果原来持有偏向锁的线程已经**死亡**，那么就直接设置为**无锁**状态，如果**没有死亡**，则代表现在是发生了锁竞争，就**升级**为轻量级锁(注意一个事，就是锁撤销的时候，会等待到安全点后暂停线程)
    - 轻量级锁获得&撤销
        - 线程复制CAS修改markword为**lock record**，即**锁记录指针**，其他线程CAS**修改失败**时，进入自旋获取的状态，如果等待很久**还是失败**，就发生锁膨胀，**主动更改**为markword状态为**重量级**锁，然后**阻塞等待**获取原有线程操作完后释放锁
- 公平锁和非公平锁的区别
    - 公平锁在并发情况下, 线程获取锁的时候, 会**查看此锁维护的等待队列**, 如果**为空或自己是队列头**就占有锁, 否则就会加入等待队列, 按照**FIFO**的规则从队列取到自己
    - 非公平锁**上来就尝试占有锁**, 占有失败再采用类似公平锁的处理方式
    - 非公平锁的**吞吐量优于**公平锁
        - 原因: 假设线程 A 持有一个锁，并且线程 B 请求这个锁。由于锁被 A 持有，因此 B 将被挂起。当 A 释放锁时，B 将被唤醒，因此 B 会再次尝试获取这个锁。与此同时，如果线程 C 也请求这个锁，那么 C 很可能会在 B 被完全唤醒之前获得、使用以及释放这个锁。这样就是一种双赢的局面，B 获得锁的时刻并没有推迟，C 更早的获得了锁，并且吞吐量也提高了
- 乐观锁
    - **总是认为不会产生并发问题**, 每次去操作数据的时候总认为不会有其他线程对数据进行修改, 因此不会上锁, 不过在**更新前会判断是否有其他线程对数据进行了修改**, 一般使用**版本号**或**CAS操作**实现
    - **并发程度低**的时候使用
    - 具体实现

```
public class AtomicInteger extends Number implements java.io.Serializable
{
  private volatile int value;
  public final int get() {
    return value;
  }
  public final int getAndIncrement() {
    for (;;) {
      int current = get();
      int next = current + 1;
      if (compareAndSet(current, next))
        return current;
```

```
    }
  }
  public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
  }
}
```

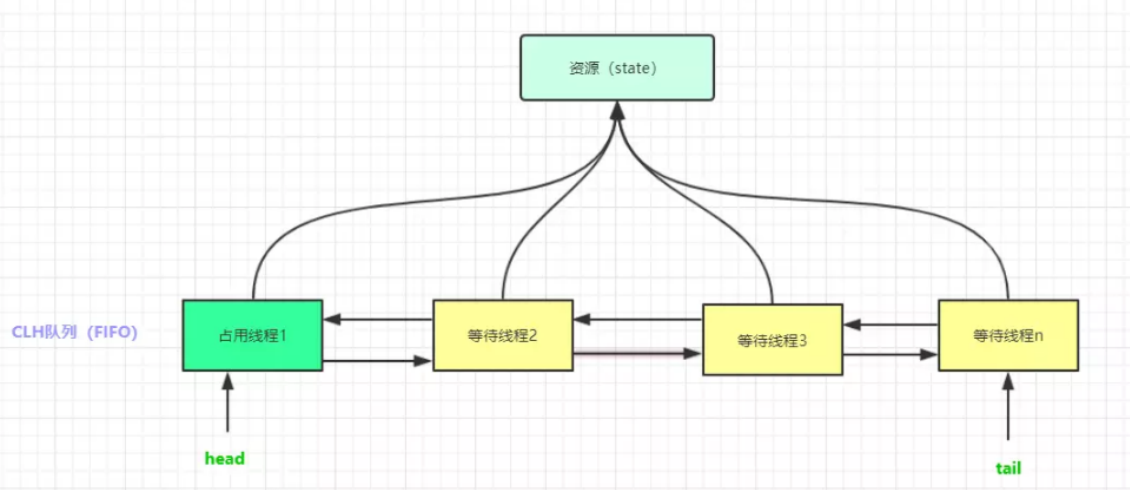
- 悲观锁
  - **总是认为会产生并发问题**, 每次去操作数据都认为其他线程会修改, 因此会**加锁**, 当其他线程想要访问数据的时候, 需要阻塞挂起
  - **并发程度高**的时候使用
- 什么是死锁?
  - 多个进程运行过程中**因竞争资源而互相等待**
- 死锁产生的4个必要条件

> https://blog.csdn.net/guaiguaihenguai/article/details/80303835

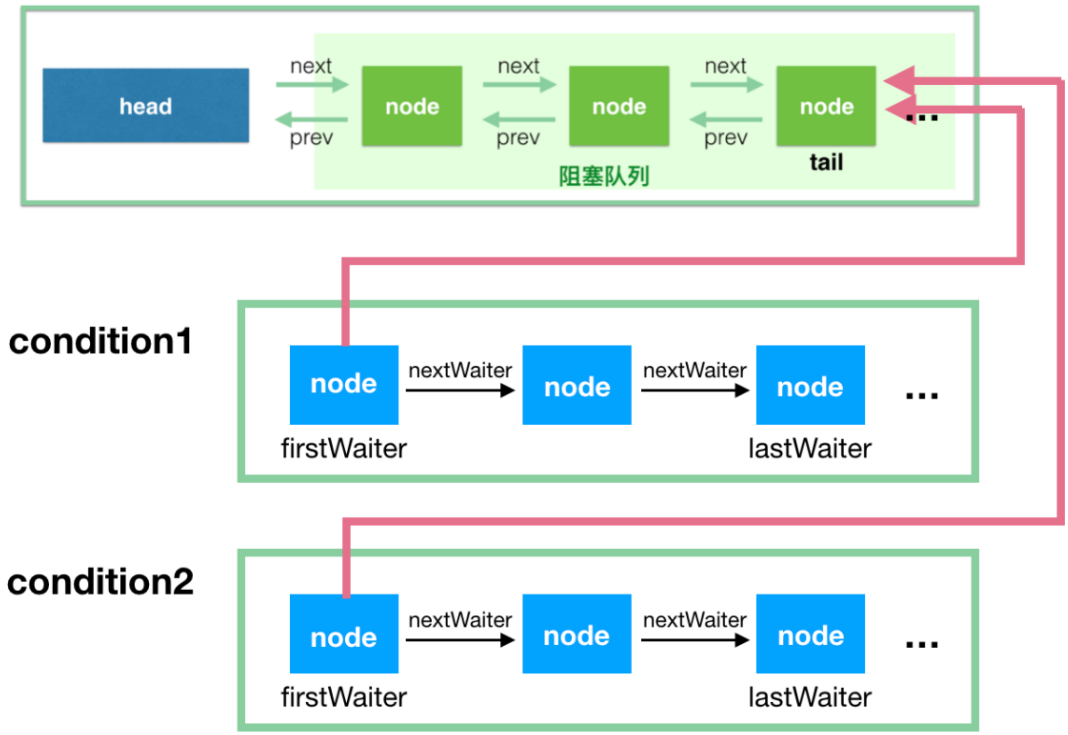
  - **互斥** : 一个资源每次只能被一个进程使用
  - **请求与保持** : 一个进程因请求资源而阻塞的时候, 不会释放已有资源
    - 解决：**一次性申请所有资源**
  - **不剥夺** : 进程已获得资源在未使用完之前不能剥夺, 只能由使用完时自己释放
    - 解决：**超时释放**
  - **环路等待** : 若干进程之间形成循环等待
    - 解决：**按序请求**
- AQS

> https://www.cnblogs.com/waterystone/p/4920797.html
>
> https://www.cnblogs.com/maratong/p/12542491.html
>
> https://juejin.im/post/5d34502cf265da1baf7d27aa
>
> AQS全称AbstractQueuedSynchronizer, 是JAVA并发包中的核心类, **ReentrantLock**和**ReentrantReadWriteLock**都是基于该类实现的

  - AQS核心思想就是基于**CLH队列**，用**volatile**修饰共享变量**state**，线程通过**CAS**去改变状态符，成功则**获取锁成功**，失败则进入**等待队列**，**等待被唤醒**
  - AQS要点
    - **state**
      - AQS内部通过state属性来代表当前**共享资源的状态**, 为0代表未被占用, 大于0代表线程持有锁
      - state使用**volatile**修饰, 保证多线程中可见性
      - getState,setState,compareAndSetState均使用final修饰, **限制AQS子类重写**
    - AQS定义了两种**资源共享方式**
      - Exclusive（独占，只有一个线程能执行，如**ReentrantLock**）
      - Share（共享，多个线程可同时执行，如**Semaphore/CountDownLatch**）
    - CLH队列
      -



- 双向队列入队出队
  - 为什么设置为双向队列?
    - 单向队列确实已经可以满足添加和移动节点了, 但是当我们判断原先在条件队列上的节点, 现在是否在阻塞队列上时, 如果从前往后遍历, 会遍历很久, 但**从后往前遍历, 就能很快确认**; 相当于用空间换时间
    - 直接进行prev是否为空的检查不行, 因为CAS操作有可能失败
  - 入队采用CAS操作, 失败自旋
  - AQS内部把每一个请求资源的线程封装成CLH队列的一个结点, head记录队首元素, tail记录队尾元素
- Condition
  -



  - 当我们调用await的时候,会进行以下几步
  - **封装当前线程为节点添加到条件队列**

        - 判断并消除已经取消的节点
        - 条件队列尾部插入节点
    - **释放原有锁/资源**
        - 获取当前state, 然后release(state)
            - 当我们没有调用lock就调用await方法时, 会在release(state)之后抛出异常
    - 当前线程挂起直到**被转移到阻塞队列**中或者被**中断**
        - 其他线程调用signal时, 会从条件队列中找到第一个需要转移的节点进行转移
    - **进入阻塞队列**, 等待获取锁
- 如何使用AQS?
    - AQS是一个已经实现了大部分方法的抽象类，我们在使用的时候只需要重写**决定如何获取锁和释放锁的方法**即可, 其余线程排队, 等待, 唤醒等操作AQS已经帮我们做好了
    - 后面解析各个AQS子类实现时，主要就是从讲解tryAcquire方法和tryRelease方法的实现来解析的
- ReentrantLock实现
    - ReentrantLock基于AQS实现, 对资源的访/问方式为**独占**, 因此主要重写了tryAcquire方法和tryRelease方法
    - ReentrantLock调用**lock**的时候, 如果是**非公平锁**, 会在直接调用**tryacquire**方法**之前**, 先调用一次**CAS**（0，1）操作尝试获取锁, 获取成功了就**直接返回**, 获取失败再和公平锁一样, 调用**tryacquire**方法
    - 由于是对一个资源的访问, 所以tryacquire方法首先获取**state**, **判断是否为0**
        - 如果**是0**的话, 直接**CAS抢锁**, 公平锁在CAS抢锁之前还需要判断**是否需要排队**, 抢锁成功就返回;
        - 如果**不为0**的话, 判断是否是**当前线程拥有锁**, 是的话说明是**重入锁**, 那么直接**set state**
    - 最后根据tryacquire方法返回值**判断是否需要加入阻塞队列**等待锁
    - ReentrantLock调用unlock的时候, 实际调用**tryrelease**方法, tryrelease中先**判断释放资源的线程和拥有锁的线程是否是同一个**, 是的话**直接setstate**, 然后返回, 再唤醒下一个节点, 不是的话就直接返回异常
- CountDownLatch

> https://blog.csdn.net/liangyihuai/article/details/83106584 代码Demo
>
> https://www.jianshu.com/p/bdf47236fc3a 原理讲解
>
> 完成的功能 : **一个线程或多个线程一直等待, 直到其他线程执行的操作完成**

    - CountDownLatch基于**AQS**实现, 核心方法为countDown和await. CountDownLatch实例**初始化**时将**count设置为AQS的state**
    - CountDownLatch主要实现了AQS的**tryAcquireShared**方法, 方法内部**getstate判断是否为0**, 当我们调用**await**方法的时候, 实际就是在不断调用tryAcquireShared方法判断是否state为0了;
    - CountDownLatch还实现了AQS的**tryReleaseShared**方法, 当我们调用**countDown**时，实际就是在调用tryReleaseShared，方法内部**CAS将state设置为state - 1**, 设置成功后再唤醒下一个节点
- CyclicBarrier
    - CyclicBarrier基于**ReentrantLock**实现,通过内部类**Generation**来代表**当前同步处于哪个阶段**, 使用**count代表剩余任务数量**, 核心方法为**await**.
    - 当线程调用await方法时, 会**对剩余任务数-1**, 然后程序会**检查当前任务是否是最后一个任务**, 如果是的话, 会**查看barrierCommand**是否为空, 不为空的话就**运行该任务**, 然后会**唤醒其他等待线程**. 如

果不是最后一个任务,当前线程会被**阻塞**, **等待最后一个任务完成**, 然后线程会通过**nextGeneration**中的**signalAll唤醒等待线程**, 等待线程被唤醒或者线程休眠超时后, 会响应中断请求, 再判断当前gengeration是否改变, 如果改变就直接返回, 进入下一轮

- CountDownLatch和CyclicBarrier区别
  - CountDownLatch和CyclicBarrier都有让多个线程等待同步然后再开始下一步动作的意思，但是CountDownLatch的下一步的动作实施者是**主线程**，具有**不可重复性**；而CyclicBarrier的下一步动作实施者还是"**其他线程**"本身，具有往复**多次实施**动作的特点
  - CyclicBarrier会**阻塞线程**，在**最后一个任务执行线程完成之前**，**其余线程都必须等待**，而线程在调用CountDownLatch的countDown方法之后就会结束
- AQS中公平锁和非公平锁的实现

> https://www.javadoop.com/post/AbstractQueuedSynchronizer-2

  - AQS内部没有提供具体方法对应公平锁和非公平锁, 只有一个统一的方法**tryacqiure**, 子类可以**重写该方法**, 一般还会再派生出两个子类, 一个代表公平锁, 一个代表非公平锁, 他们内部会提供相应方法
  - 但总体区别可以认为是 在**当前有资格获取锁的时候**, **公平锁**会调用父类的**hasQueuedPredecessors**方法判断**是否队列为空**或者**自己是队列头**, 判断**成功后再CAS获取锁**, 但是**非公平锁直接CAS获取锁**
    - CAS获得锁失败后, 就会把自己**加入到阻塞队列的尾部**（这里的尾部实际上是从后往前找第一个未被取消的节点/安全点），然后调用LockSupport.park(this)使线程进入等待状态，每次被唤醒(unpark()/interrupt())后，会**判断自己是否到达了队列头**, **如果到达了队列头**, 再**调用try acquire继续尝试**，否则继续进入park状态
    - 队列中的每一个元素都会不断检查自己是否到达了队列的头部, 然后尝试调用try acquire
  - **ReentrantLock**实现的时候, 非公平锁在调用lock之后, 还会**提前CAS抢锁一次**, 再进入tryacquire方法
- AQS中读写锁的实现

> https://zhuanlan.zhihu.com/p/91408261
>
> 两个int实现简易读写锁 : https://www.cnblogs.com/DarrenChan/p/8619476.html

  - AQS中读写锁使用一个state变量来表示读状态和写状态, **高16位是读**, **低16位是写**
  - 写锁
    - 调用写锁的**lock**时, 会调用读写锁实现的tryAcquire方法.
    - tryAcquire中, 首先会获得state
      - 如果**state为0**, 则说明没有人读也没有人写, 那么就**直接CAS设置锁**, 如果是**公平锁**, 那么在CAS设置锁之前还需要**进行队列是否有节点的判断**.
      - 如果**state不为0,** 判断是否**有线程在读**或者**有线程在写但是写线程是自己**, 然后判断是否**要设置的state超过最大值**, 如果**均通过**, 就直接**set state返回** （不能锁升级）
    - 如果**tryAcquire获取失败**了, 说明当前线程**不具有获取锁的资格**, 那么就把当前线程封装成节点**加入等待队列**里面, 调用park让自己进入等待状态，然后不断判断是否到达了队列头,如果到达了再进行tryAcquire
    - 调用写锁的**unlock**时,会调用子类实现的tryRelease, tryRelease内部直接进行**set state**; tryRelease完后会尝试唤醒下一个节点

        - 读锁
            - 调用读锁的**lock**时, 会调用读写锁中实现的tryAcquireShared方法.
                - tryAcquireShared方法中, 首先会获得state, 判断是否**有人在写或者写的线程是自己, 如果有人在写且不是自己的话就会直接返回** （可以锁降级）
                - 这时候其实就可以开始尝试**CAS获得锁**了, 但在CAS设置锁之前, 还需要**判断是否需要排队**(公平锁)
                - 如果不需要判断且获得锁成功了, 就**更新统计的读锁次数**(存储在**ThreadLocal**)
                - 如果失败了的话就调用**fullTryAcquireShared**方法, **不断的CAS去获取锁**
            - 如果调用**tryAcquireShared失败**了, 同样会把当前线程**加入等待队列**, 调用park等待， 被唤醒后不断判断是否是队列头
            - 调用读锁的**unlock**时, 会调用子类实现的tryReleaseShared, tryReleaseShared内部**直接进行set state**; tryReleaseShared完后会尝试唤醒下一个节点
        - 写锁的饥饿是如何解决的?
            - java.util.concurrent.locks.ReentrantReadWriteLock.NonfairSync#readerShouldBlock
            - 写锁发现有人在读之后, 就把自己加入阻塞队列
            - 当后续**读锁想要获得读锁的时候**, 就会检查阻塞队列里面的**第一个等待节点是不是在等待写锁**, 如果是的话, 就**把自己加入到阻塞队列**里面去
        - 注意
            - 读写锁不支持**锁升级**(会出现死锁), 支持**锁降级**(保证同一线程可见性)
            - 读锁不支持条件队列, **条件队列适合用于唤醒指定的线程去获取锁**, 但是读锁是大家**可以同时获得**的, 所以没必要使用条件队列
    - Semaphore实现

        > https://www.cnblogs.com/doit8791/p/9163634.html
        >
        > https://blog.csdn.net/qq_37142346/article/details/80344996
        >
        > https://www.cnblogs.com/tong-yuan/p/Semaphore.html

        - 首先Semaphore是基于AQS实现的, 当我们初始化Semaphore的时候, **传入的permits会被赋值给state**, 代表**当前剩余共享资源的数量**
        - 当我们调用acquire方法的时候, 首先会调用**tryAcquireShared**方法**CAS**的尝试**去获得资源**, 如果**资源不够**的话, 就会**把当前线程加入阻塞队列**, 然后不断的检查自己是否到了队列首, 如果到了的话就会继续尝试获得资源, 不够的话再继续重复这个过程
        - 当我们调用release方法时,首先会调用**tryReleaseShared**方法**CAS**的尝试去**增加资源**, 操作完之后; 会尝试去**唤醒后面的节点**(全部)
- ThreadLocal

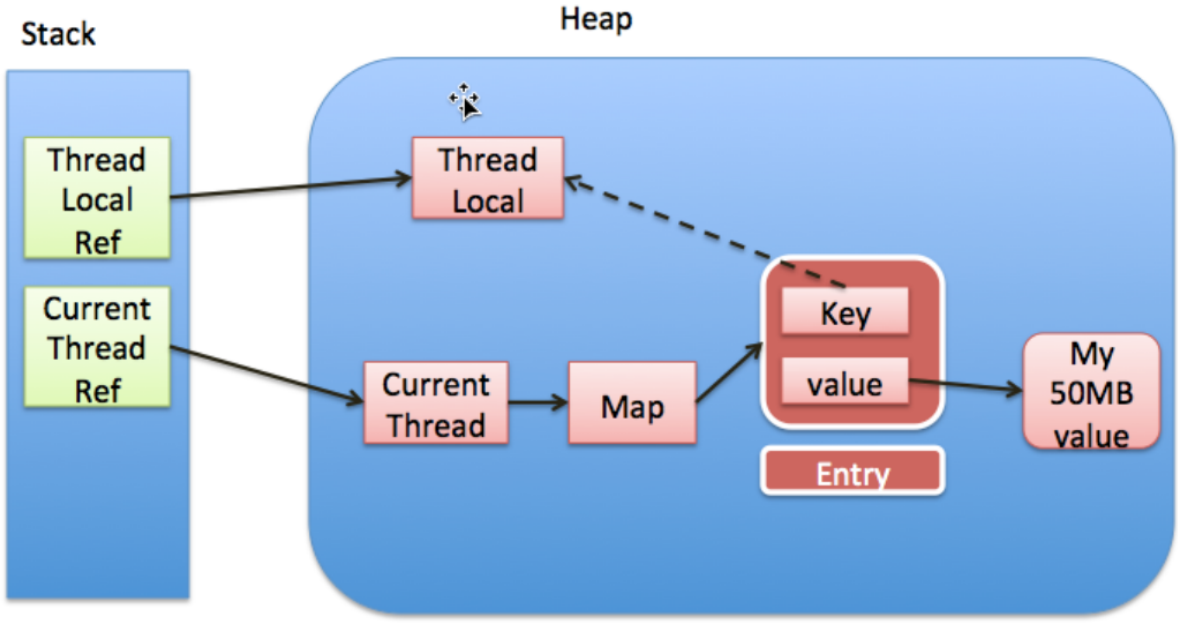
    > https://www.jianshu.com/p/dc9be75b8efd
    >
    > 实现**相同线程数据共享, 不同线程数据隔离**

    - 实现原理 :
    - ThreadLocal对象进行**set**的时候, 会**绑定**一个**ThreadLocalMap**到线程上并获取该ThreadLocalMap, 当然如果**已经绑定过了就不再进行绑定**, 获取到ThreadLocalMap后, 再使用该ThreadLocalMap来完成set操作, 即**值真正是放在ThreadLocalMap 中存取**的, **ThreadLocalMap 内部类有一个Entry 类， key是ThreadLocal 对象, value 就是你要存放的值**, 这样就实现了把ThreadLocal变成线程本地变量

    - 开放地址法
        - ThreadLocalMap 的数据结构**仅是数组**, 采用**开放地址法**来解决hash冲突
        - 开放地址法和链地址法比较
            - 开放地址法容易产生**堆积**问题, **不适合大规模数据存储**
            - 开放地址法的**散列函数设计得不好**的话**容易产生多次冲突**
            - **删除**的元素如果是**冲突元素**中的一个, 需要**对后面的元素作处理**,实现较为复杂
            - 链地址法**不容易产生堆积**, **平均查找长度小**, **删除易实现**
            - **综上** : 规模小用开放地址法, 否则用链地址法
    - 弱引用
        - 由于ThreadLocalMap中的key-ThreadLocal 是弱引用, 而value是强引用, 因此在key的生命周期结束后就发生GC就会被回收, 这样就有可能会造成内存泄漏, 而针对弱引用, ThreadLocalMap的**set方法也进行了特殊的处理**
        - 进入set方法, 首先**获取索引**, 然后**依次遍历**, 遇到key**相同就直接覆盖返回**, 遇到**key为null**则说明发生了**ThreadLocal被回收**的情况, 那么就调用**replaceStaleEntry**方法进行替换, 遍历到**数组为null**仍未出现key相同或被回收, 就**直接插入**
        - 进入replaceStaleEntry方法, 会**对无用entry进行回收**, 回收的时候会顺便**把前面的无用entry也回收**掉, 因此会**往前找连续entry中最早出现的无用entry**, 然后再**往后找相同的key的位置**, 进行交换并赋新值, 防止相同key出现两次, 然后调用expungeStaleEntry方法传入最早出现的无用entry标记尝试进行无用entry的清除
        - 进入expungeStaleEntry方法后, 会一直进行遍历直到null为止, 途中**遇到无用entry就设置为null,** 方便gc回收, 遇到**有用entry就使用开放地址法重新确认其位置**
    - 内存泄漏问题

      > https://blog.csdn.net/buyulian/article/details/103465615
      > https://blog.csdn.net/qq_27127145/article/details/83868085

        - 通过上面的分析，我们知道expungeStaleEntry() 方法是**帮助垃圾回收**的，根据源码，我们可以发现 **get 和set 方法**都可能**触发清理**方法expungeStaleEntry()，但是这些清理不是必须发生的，因此，从某种角度来说，**可能会长时间存在key为null**，**value不为null的数据**，可以认为发生了内存泄漏，因此，我们应该养成好习惯**不再使用的时候调用remove()**, 加快垃圾回收, 避免内存泄漏
- 为什么Thread中的变量定义是ThreadLocalMap而不是ThreadLocal?
    - ThreadLocal仅仅是一个**代理工具类**, 内部不持有任何与线程相关数据, 所有与线程相关数据存储在Thread里面容易理解, 其次, 这样设计不容易产生内存泄漏, 因为如果Thread中的变量定义是ThreadLocal, 那么只要ThreadLocal存在, Thread就不会被回收, 而往往**ThreadLocal的生命周期比Thread更长**, 这样就**容易导致内存泄漏**
- ThreadLocal为啥设计为弱引用?



    - ThreadLocal如果**设计为强引用**, 在我们把ThreadLocal**设置为null后**, **由于ThreadLocal还被ThreadLocalMap中的entry强引用**着, 导致其**无法被回收**, 造成**内存泄漏**, 除非线程结束；其实弱引用只是减少了部分情况的内存泄漏，在某些情况下还是会产生内存泄漏(比如多线程，方法级别创建ThreadLocal，set、get没有触发清理操作)
  - ThreadLocal是存在 jvm 内存哪一块的?
    - **堆**, 对象都是存储在堆里面的
  - 应用场景
    - **Spring**使用ThreadLocal**解决线程安全问题**, 将线程不安全的变量放到ThreadLocal中（由于单例, 所以可以认为没有内存泄漏问题）

线程&线程池

线程

- 进程和线程的区别?

  > https://www.cnblogs.com/zhehan54/p/6130030.html

  - 进程是程序的**一次执行过程**, 是系统**运行程序**的**基本单位**, 系统运行一个程序就是**一个进程从创建, 运行到消亡的过程**; 在java中, 当我们启动**main函数**的时候, 其实就是启动了一个**JVM进程**, 而main就是这个进程内的一个线程, 也称**主线程**
  - 线程和进程相似, 一个进程在执行过程中可以**产生多个线程**, 与进程不同的是, 同类线程间**共享**进程的**堆**和**方法区**资源, 但每个线程有**自己**的**程序计数器**, **虚拟机栈**和**本地方法栈**, 所以系统在创建线程或者切换线程的时候, 消耗的资源比进程小得多, 因此线程也被称为**轻量级进程**
  - 因此进程和线程的区别如下：
    - 进程是**资源分配**的最小单位，线程是**程序执行**的最小单位
    - 进程有自己的**独立地址空间**，每启动一个进程，程序就会为它分配地址空间，建立**数据表**来维护代码段，堆栈段和数据段，这种**操作非常昂贵**。而线程是共享进程中的数据的，使用**相同的地址空间**，因此CPU切换一个线程的开销是要比切换一个进程的**开销小很多**的，同样，创建一个线程的开销也要比创建一个进程的开销小很多
    - 线程间的**通信**比进程间的通信更加方便，因为同一进程下的线程**共享全局变量**，**静态变量**等数据，

而进程间的通信需要以**通信**的方式进行，不过如何处理好同步与互斥也是编写多线程程序的难点。
        - ~~多进程程序比多线程程序更加**健壮**，因为多线程程序只要有一个线程死掉，那么整个进程也就死掉了，但是多进程程序中一个进程死掉不会对另一个进程产生影响，因为进程有自己独立的地址空间~~
- Thread源码理解

    > https://www.jianshu.com/p/81a56497e073

    - 简单来说就是，调用start方法后，主要是调用一个start0的**native方法**去**申请相关线程资源**，然后在native方法中申请到资源后，就会**调用run方法**
- 谈谈对多线程的理解
    - 使用多线程的好处
        - 多线程**充分利用了计算机处理性能**, 提高程序运行效率 (单核提高了CPU和IO设备的综合利用率)
        - 多核情况下, 单线程只会利用一个CPU, 多线程可以**同时利用多个CPU**
        - 可以**提高并发能力**
    - ~~怎么实现多线程~~
        - ~~同实现线程的方法~~
    - 使用多线程带来的问题
        - 线程不安全
        - 资源消耗
        - 死锁
    - 怎么解决
        - ThreadLoacl, 加锁, CAS
        - 线程池
        - 检查逻辑, 修改代码
- 什么是上下文切换?
    - 当前任务执行完CPU时间片切换到另一个任务之前, 会先**保存自己的状态**, 以便下次再切换为这个任务时, 可以**再次加载**这个任务的状态
- 什么时候会发生上下文切换?
    - 线程**CPU时间用完**
    - **GC垃圾回收**
    - 有**更高优先级线程**运行
    - 线程调用了一些**让出CPU使用权**的方法
- 并行和并发的区别?
    - 并发是多个事件在**同一时间间隔**发生, 是**交替**的
    - 并行是多个事件在**同一时刻**发生, 是**同时**的
- 了解协程么?

    > https://www.liaoxuefeng.com/wiki/897692888725344/923057403198272

    - 子程序运行过程中的中断
- 线程间通信方式
    - **共享内存**和**消息传递**
    - 以下方式均是基于上面两种模型实现
        - **volatile**

```
package thread;
public class MyThreadTest {
  public static void main(String[] args) throws Exception {
    notifyThreadWithVolatile();
  }
  //定义一个测试
  private static volatile boolean flag = false;
  //计算I++，当I==5时，通知线程B
  private static void notifyThreadWithVolatile() throws Exception {
    Thread thc = new Thread("线程A"){
      @Override
      public void run() {
        for (int i = 0; i < 10; i++) {
          if (i == 5) {
            flag = true;
            try {
              Thread.sleep(500L);
            } catch (InterruptedException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
            }
            break;
          }
          System.out.println(Thread.currentThread().getName() + "====" 
+ i);
        }
      }
    };

    Thread thd = new Thread("线程B") {
      @Override
      public void run() {
        while (true) {
          // 防止伪唤醒 所以使用了while
          while (flag) {
            System.out.println(Thread.currentThread().getName() + "收到
通知");
            System.out.println("do something");
            try {
              Thread.sleep(500L);
            } catch (InterruptedException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
            }
            return ;
          }

        }
```

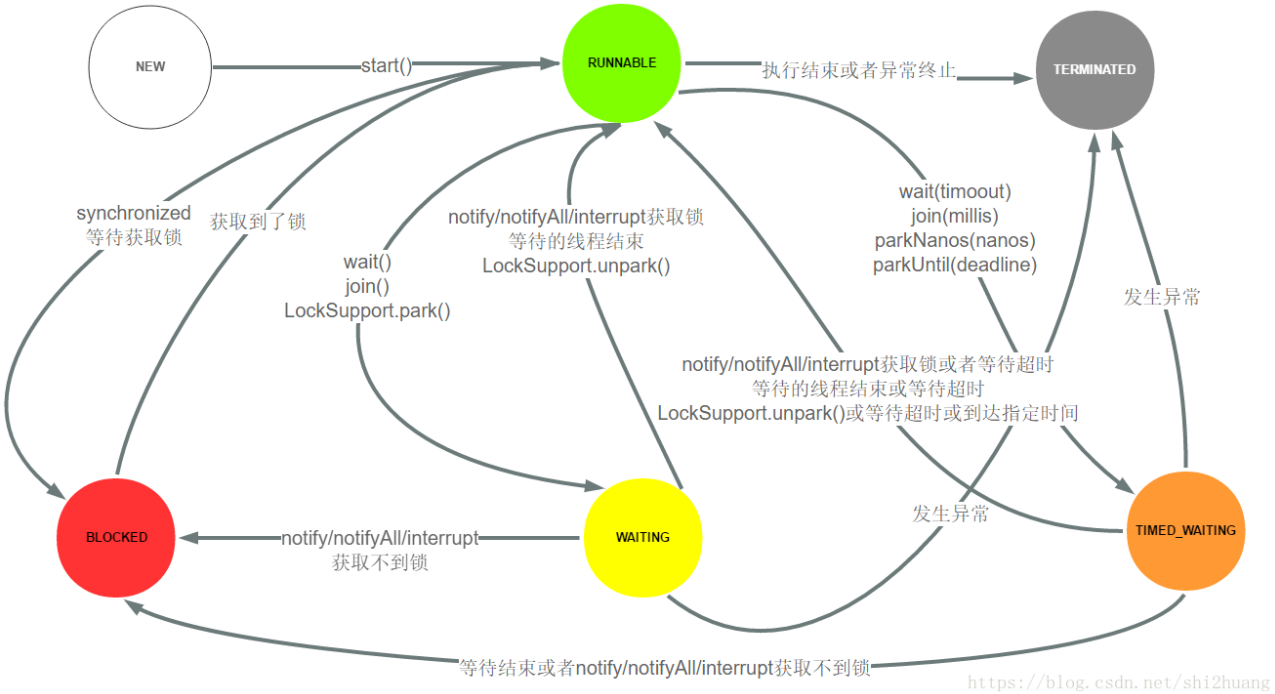
```
        }
      };
      thd.start();
      Thread.sleep(1000L);
      thc.start();
    }
}
```

  - **锁**
  - **ThreadLocal**

- 线程的生命周期有哪些状态？怎么转换？

> https://www.cnblogs.com/waterystone/p/4920007.html
> https://blog.csdn.net/shi2huang/article/details/80289155

> 注意**阻塞和等待的转换**，比较有意思



  - 阻塞和等待的区别
    - 阻塞：当一个线程试图**获取对象锁**（非java.util.concurrent库中的锁，即**synchronized**），而该锁被其他线程持有，则该线程进入阻塞状态；它的特点是**使用简单，由JVM调度器来决定唤醒自己，而不需要由另一个线程来显式唤醒自己，不响应中断**
    - 等待：当一个线程等待另一个线程通知调度器一个条件时，该线程进入等待状态。它的特点是**需要等待另一个线程显式地唤醒自己，实现灵活，语义更丰富，可响应中断**。例如调用：Object.wait()、Thread.join()以及等待Lock或Condition

> 需要强调的是虽然synchronized和JUC里的Lock都实现锁的功能，但线程进入的状态是不一样的。**synchronized会让线程进入阻塞态，而JUC里的Lock是用LockSupport.park()/unpark()来实现阻塞/唤醒的，会让线程进入等待态**。但话又说回来，虽然等锁时进入的状态不一样，但被唤醒后又都进入runnable态，从行为效果来看又是一样的

- wait&sleep
  - wait 和 sleep 有什么区别?
    - wait和sleep方法都可以用来**放弃CPU执行权**, 当线程**持有锁**的时候, sleep方法**不会放弃锁**, wait方法**会放弃锁**
  - 什么情况下会用到 sleep?
    - 需要**减慢线程执行**的时候
- 怎么停止线程?

  > https://www.cnblogs.com/liyutian/p/10196044.html

  - 使用**标识符**
  - 使用**stop**方法, 不推荐使用, 因为是强行中止, finally语句不执行, 直接释放所有锁
  - 使用**interrupt**方法, 通知目标线程有人希望你终止, 需**自定义中断代码**
- 怎么控制多个线程按序执行?
  - **锁+标识符**
  - **join方法**

    > https://www.cnblogs.com/huangzejun/p/7908898.html

    - 当我们调用某个线程的这个方法时，这个方法会**挂起调用线程**，直到**被调用线程结束执行**，调用线程才会继续执行
    - 实现原理：while&wait&notifyAll
  - **单线程池(单队列)**
- 线程创建方法?
  - **继承 Thread 类** (Thread类本质上是实现了Runnable接口的一个实例)
  - **实现Runnable接口**
  - **实现Callable接口**(带有返回值)
    - Callable和Runnable区别
      - Callable有返回值, Runnable没有
      - Callable只能通过线程池提交
      - Callable有容错机制, 异常可以抛出, Runnable没有
  - 基于**线程池**

线程池

- 为什么要使用线程池?
  - 线程**可重用**, **减少创建销毁开销**
  - 可**控制最大并发线程数**, **避免过多竞争**
  - 可提供**定时**, **定期**执行, **单线程**执行任务等功能
- 线程池有哪几种?
  - newCachedThreadPool : 复用以前可用的, **不够就新添**, **长期不用的移除**
    - 使用场景 : **耗时较短**的任务
    - 具体参数：`0, Integer.MAX_VALUE, 60s, SynchronousQueue`
    - 考虑问题：https://blog.csdn.net/wenniuwuren/article/details/51700080

    - newFixedThreadPool : **固定线程数**, 可复用, **不够就加入等待队列**
        - 使用场景 : 需要**控制并发运行线程数量**的任务
        - 具体参数: `nThreads, nThreads, 0, LinkedBlockingQueue`
    - newScheduledThreadPool : 创建可**定时执行**或**定期执行**的线程池
        - 使用场景 : 执行定时定期任务
        - 具体参数: `corePoolSize, Integer.MAX_VALUE, 0, DelayedWorkQueue`
    - newSingleThreadExecutor : 一个线程死亡后重启另一个线程执行
        - 使用场景 : 按顺序执行线程
        - 具体参数: `1, 1, 0, LinkedBlockingQueue`
- ThreadPoolExecutor 有什么参数? 各有什么作用? 拒绝策略?

    > https://www.cnblogs.com/yefeng654321/articles/11253842.html

    ```
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory,
                              RejectedExecutionHandler handler)
    ```

    - corePoolSize : 线程池**核心线程数**, 即使线程空闲也会一直保留
    - maximumPoolSize : 线程池**最大线程数**
    - keepAliveTime : **空闲线程存活时间**, 超过核心线程数的线程在空闲这么多时间后会被移除
    - unit : **时间单位**
    - workQueue : **缓冲队列**, 存储**等待执行的任务**, 线程池无可用线程时加入队列
        - SynchronousQueue（默认）：直接提交，如果没有线程可以处理，那么就直接失败
        - ArrayBlockingQueue
        - LinkedBlockingQueue
        - ~~LinkedBlockingQueue和ArrayBlockingQueue主要区别~~
            - ~~ArrayBlockingQueue采用数组实现, LinkedBlockingQueue采用链表实现~~
            - ~~ArrayBlockingQueue初始化需要传入队列大小, LinkedBlockingQueue可以不传入~~
            - ~~ArrayBlockingQueue用一把锁控制并发, LinkedBlockingQueue用两把锁控制并发~~
    - threadFactory : **线程工厂**, 用于**创建线程**
    - handler : 队列已满,且任务量大于最大线程数的**异常处理策略**
        - **丢弃任务并抛出拒绝异常** (默认)
        - **丢弃任务不抛出异常**
        - **丢弃最前面**的任务, 并**尝试重新执行**
        - 在**调用者线程**中**提交**该任务 https://www.jianshu.com/p/9fec2424de54
- 一个任务从被提交到被执行，线程池做了哪些工作?
    - 线程池线程数小于corePoolSize, 即使线程空闲, 也要**创建新线程**处理任务
    - 线程池线程数等于corePoolSize, 且缓冲队列未满, 则把任务**放入缓冲队列**

    - 缓冲队列已满, 线程池线程数大于corePoolSize但小于maximumPoolSize, **创建新线程**处理任务
    - 缓冲队列已满, 线程池线程数大于maximumPoolSize, 通过handler指定的**拒绝策略**来处理任务
    - 总结 : 处理顺序为 : 核心线程池的线程 > 缓冲队列 > 线程池的线程 > 拒绝策略
- 线程复用原理

> https://blog.csdn.net/vincent_wen0766/article/details/108533637
> getTask()内部通过检查配置来判断是否需要在CAS获取任务时，进行阻塞/超时返回

    - 线程复用原理在源码中的runWorker方法可以体现, 主要逻辑就是线程**除了执行创建时指定的任务外**, 还会**主动**从任务队列中**取出任务执行**, 如果没取到的话会从阻塞队列里面获取任务，阻塞队列没有的话，就**阻塞等待**（阻塞队列内部用AQS实现，所以实际也是利用LockSupport.park(this)来实现的阻塞）, 从而**保证线程的存活**
- CTL属性
    - 高3位代表**线程池状态**
    - 低29位代表**线程池存活线程数**
- 线程池5种状态
    - RUNNING
        - 这个状态代表线程池可以正常判断新提交的任务, 也可以处理缓冲队列中的任务
    - SHUTDOWN
        - 这个状态代表线程池不能接收新提交的任务, 但可以处理缓冲队列中已保存的任务, 可以由RUNNING状态调用shutdown()转到该状态
    - STOP
        - 这个状态代表线程池不能接收新提交的任务, 也不能处理缓冲队列中已保存的任务, 并且能中断现在线程中的任务. 可以由RUNNING和SHUTDOWN状态调用shutdownNow()方法进入该状态
    - TIDYING
        - SHUTDOWN状态下缓冲队列为空, 且工作中的线程数为0会进入该状态, STOP状态下工作中的线程数为0也会进入该状态
    - TERMINATED
        - TIDYING状态下调用terminated()进入该状态, 可以认为该状态是最终的终止状态
- CPU密集型任务线程池参数选择

> https://blog.csdn.net/sxllllwd/article/details/100533788

    - 特点:需要大量计算, 主要消耗CPU资源, 任务越多, 花在任务切换的时间就越多, CPU执行任务的效率就越低, 因此应**配置核心线程数为CPU的核心数**, 可以选择newFixThreadPool
- IO密集型任务线程池参数选择

> https://zhuanlan.zhihu.com/p/94679167

    - 特点:需要执行大量IO操作, 对CPU消耗较少, 因此应**配置尽可能多的线程**, 比如2*CPU数目, 最好自己拓展线程池, 不使用原生的, 因为原生线程池适合于CPU密集型任务