* 学习目标

  * 能够掌握多线程之并发工具类

    * CountDownLatch:计算器，减，countDown

    * CyclicBarrier:计算器,加

    * Semaphore:信号灯，限定线程数量

    * LockSupport：park，unpark

  * 能够理解Java内存模型

    * Java Memory Model:JMM

    * 内存访问跨平台，变量访问规则，共享变量

    * 原子性，可见性，有序性

    * 主内存，工作内存

      * 8操作：主lock-主read-工load-工use-工assign-工store-主write-主unlcok

      * 8个规则：主变量-->工不改不回主->一线lock->lock清工-->unlock-->unlcok--readload,storewrite-->assgin

  * 能够理解指令重排序

      * 优化性能（编译器和CPU）

      * 单线程：as-if-serial,数据依赖

      * 多线程下：禁用指令重排序volatite

  * 能够理解多线程的内存可见性

      * volatite.加锁，原子类

  * 能够理解线程安全性CAS

    * CAS: Compare And Swap

        * 内存地址V里面值，预期值A，需要更新值B

          * 确保线程安全：当V里面值和预期值想，就V里面值更新B

      * atomic: CAS

* 能够掌握BlockingQueue相关阻塞队列

    * BlockingQueue:阻塞，生产者和消费者模式

        * put,take

        * ArrayBlockingQueue: 有边界

    * ...

---

* 回顾

  * 线程安全的问题：多个线程同时去访问一个共有变量，可能会出现线程安全问题

    * 卖电影票

  * 加锁

    * synchronized(lock)

        * 代码块，方法

    * ReentrantLock

        * lock.lock(),lock.unlock()

  * ReentrantLock 额外的功能

    * 实现公平锁，性能较差

        ReentrantLock lock=new ReentrantLock(true);

    * 可中断，解决死锁问题

        * 死锁：互斥，占有且等待，不可抢夺，循环等待

        * lockInterruptBy()

    * tryLock,解决死锁问题

    * Condition,await,singal,singalAll\

  * 生成者和消费者模式

    生产者--->缓存区--->消费者

    * 加锁

    * 等待和唤醒机制

        * synchronized+wait,notify,notifyAll

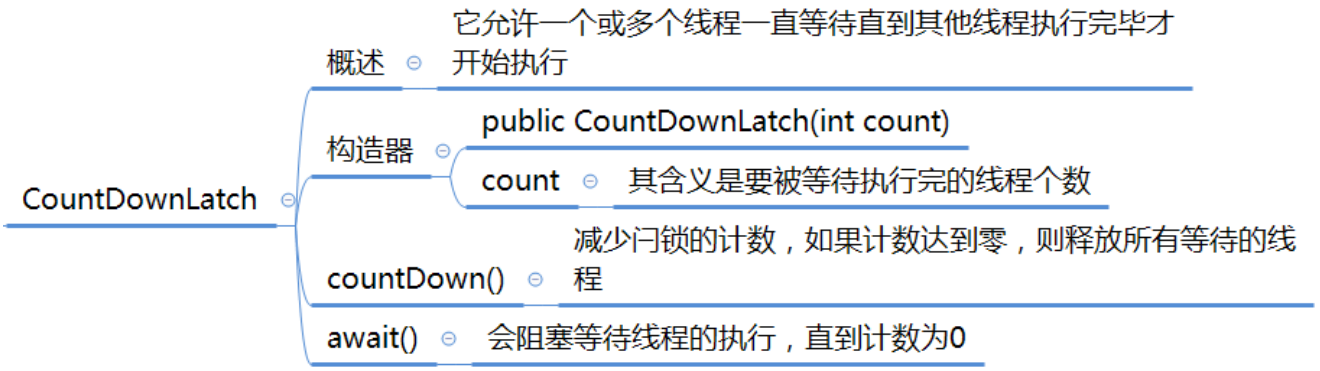
        * ReentrantLock-lock,unlock,condition-await,singal,singalAll

```
}
```

* 能够掌握多线程之并发工具类

    * JDK1.5引入了concurrent并发包，常见的并发工具类

* CountDownLatch




```
* 需求：多个线程同时工作，然后其中几个可以随意并发执行，
  但有一个线程需要等其他线程工作结束后，才能开始。
  例如：开启多个线程分块下载一个电影，每个线程只下载固定的一截，
  最后由另外一个线程来拼接所有的分段

```

```
* 代码
public class Main {
    // 3个要被等待执行完的线程个数
    private static CountDownLatch latch=new CountDownLatch(3);
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName()+"线程开始下载
                try {
                    latch.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+"线程下载完成
                System.out.println(Thread.currentThread().getName()+"线程合并文件
            }
        },"download").start();
        for (int i = 0; i < 3; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    // 模拟线程下载
                    System.out.println(Thread.currentThread().getName()+"线程开始
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()+"线程下载
                    latch.countDown();
                }
            },"t"+i).start();
        }
    }
}

* 结果
download线程开始下载...
t0线程开始下载...
```

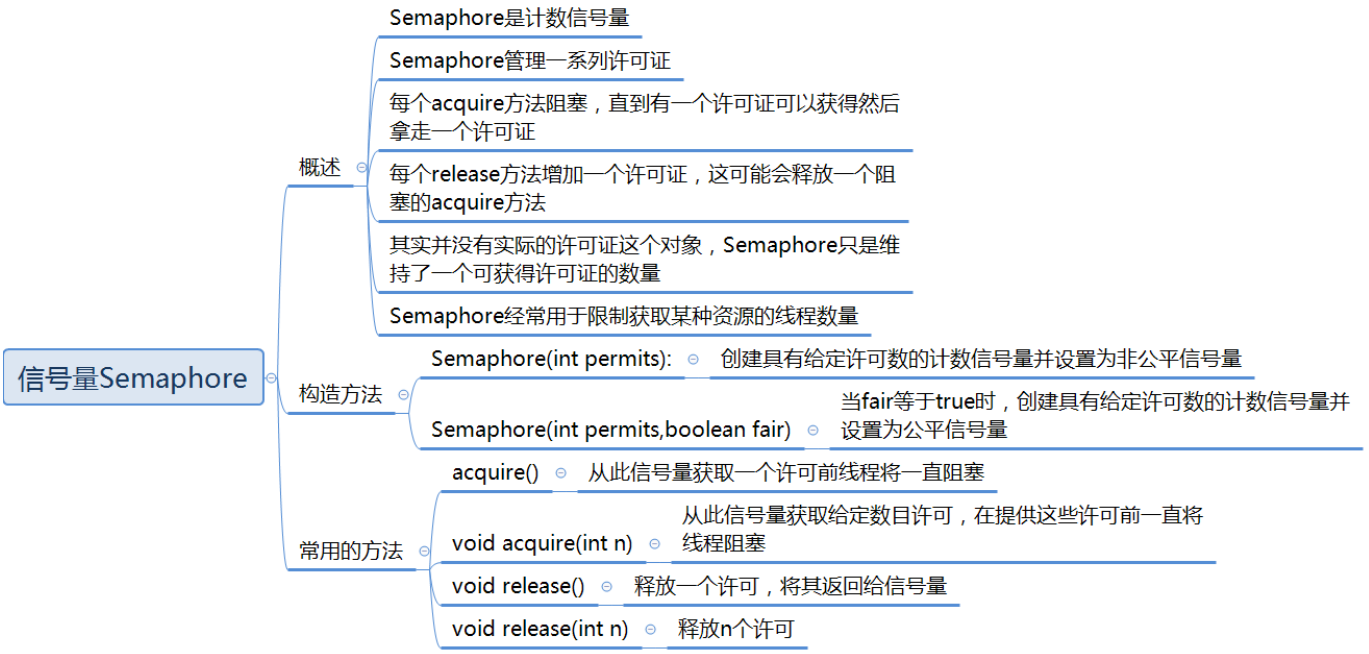
```
t1线程开始下载...
t2线程开始下载...
t1线程下载完成...
t0线程下载完成...
t2线程下载完成...
download线程下载完成...
download线程合并文件
```

* CyclicBarrier


```
* 需求：开启多个线程分块下载一个电影，每个线程只下载固定的一截，
  最后所有线程下载完成了，触发方法来拼接所有的分段
* 代码
public class Main {
    public static void main(String[] args) {
        // 3个要被等待执行完的线程个数
        CyclicBarrier cyclicBarrier=new CyclicBarrier(3, new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName()+"线程合并文件
            }
        });

        for (int i = 0; i < 3; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    // 模拟线程下载
                    System.out.println(Thread.currentThread().getName()+"线程开始
```

```
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()+"线程下
                   // 等待所有 parties已经在这个障碍上调用了 await
                    try {
                        cyclicBarrier.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } catch (BrokenBarrierException e) {
                        e.printStackTrace();
                    }
                }
            },"t"+i).start();
        }
    }
}
* 结果
t0线程开始下载...
t1线程开始下载...
t2线程开始下载...
t0线程下载完成...
t2线程下载完成...
t1线程下载完成...
t2线程合并文件

```

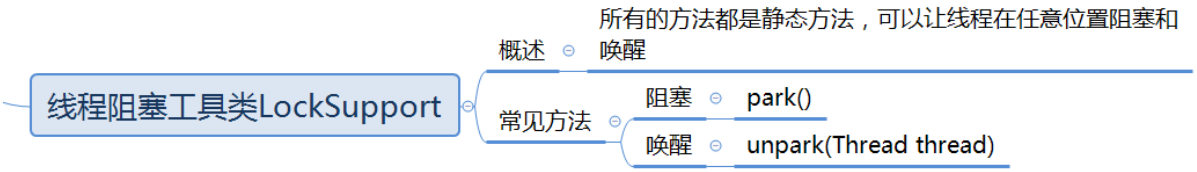
* Samaphore




```
* 案例：
 * 停车场上有5个停车位，一个停车位一次只能停一辆车，
 一旦所有停车位在使用，那么后面的来的车就需要等待，直到有一辆车开出某个停车位
 后面来的车才能进来
* 代码实现
public class Main {
    public static void main(String[] args) {
        // 5个停车位
        Semaphore semaphore=new Semaphore(5);
        // 10辆车近来
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    // 获取（许可证）车位
                    try {
                        semaphore.acquire();
                        System.out.println(Thread.currentThread().getName()+"线
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //模拟停车的时间
```

```
                    try {
                        Thread.sleep(5000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // 释放许可证：车开走
                    semaphore.release();
                    System.out.println(Thread.currentThread().getName()+"线程释放
                }
            },"t"+i).start();
        }
    }
}
```

* LockSupport

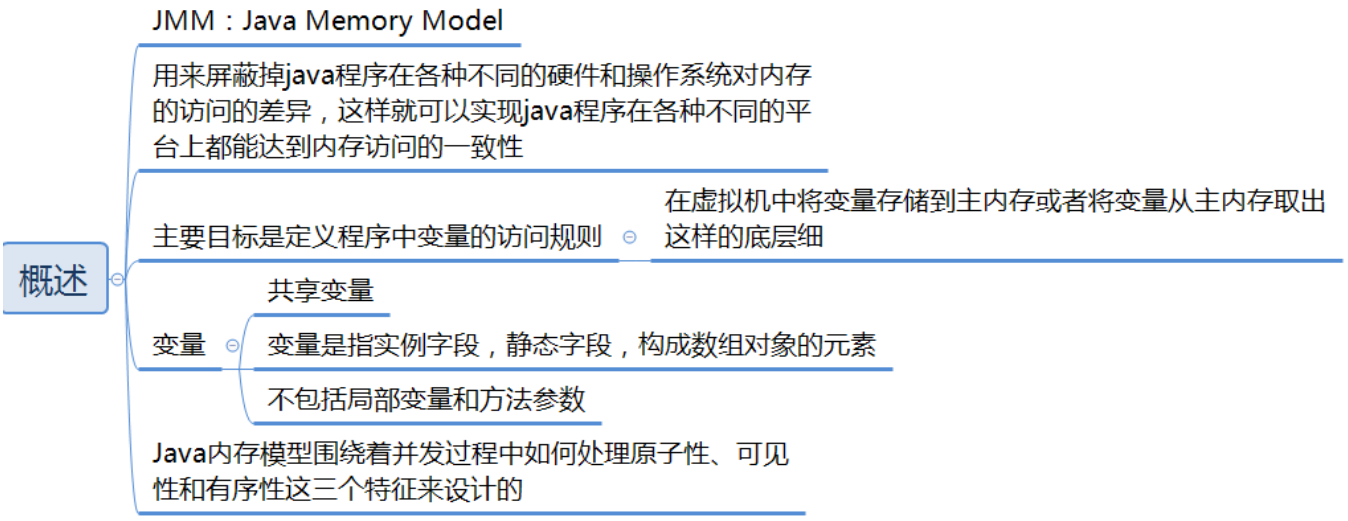



```
* 案例
public class Main {
    public static boolean isFinish=false;
    public static void main(String[] args) {
        // 显示图片的线程
        Thread showImgThread=new Thread(new Runnable() {
            @Override
            public void run() {
                // 等待图片下载完
                LockSupport.park();
                if(!isFinish){
                    throw new RuntimeException("show:图片还没有下载完");
                }
                System.err.println("show:图片显示完成！");
            }
        });
```

```
        // 下载图片的线程
        Thread downThread=new Thread(new Runnable() {
            @Override
            public void run() {
                // 模拟开始下载
                System.out.println("download:开始下载图片...");
                for (int i = 1; i <= 100; i++) {
                    System.out.println("download:已完成" + i + "%");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("download:图片下载完毕");
                isFinish = true;
                LockSupport.unpark(showImgThread);
                System.out.println("download:开始下载视频...");
                for (int i = 1; i <= 100; i++) {
                    System.out.println("download:已完成" + i + "%");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        downThread.start();
        showImgThread.start();

    }
}
```

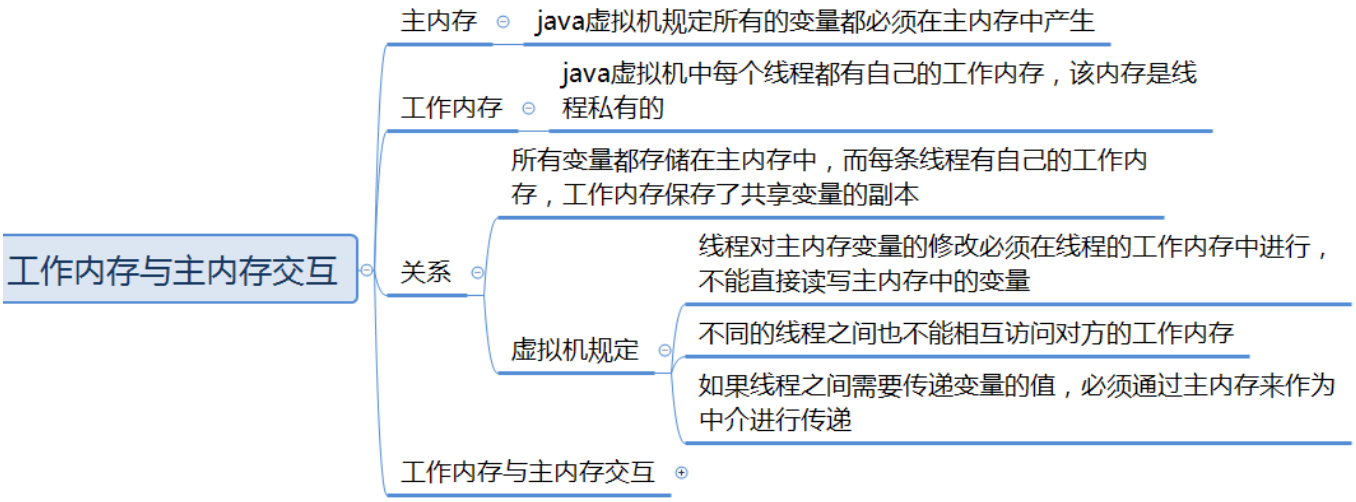
* 能够理解Java内存模型

  * Java内存模型概述

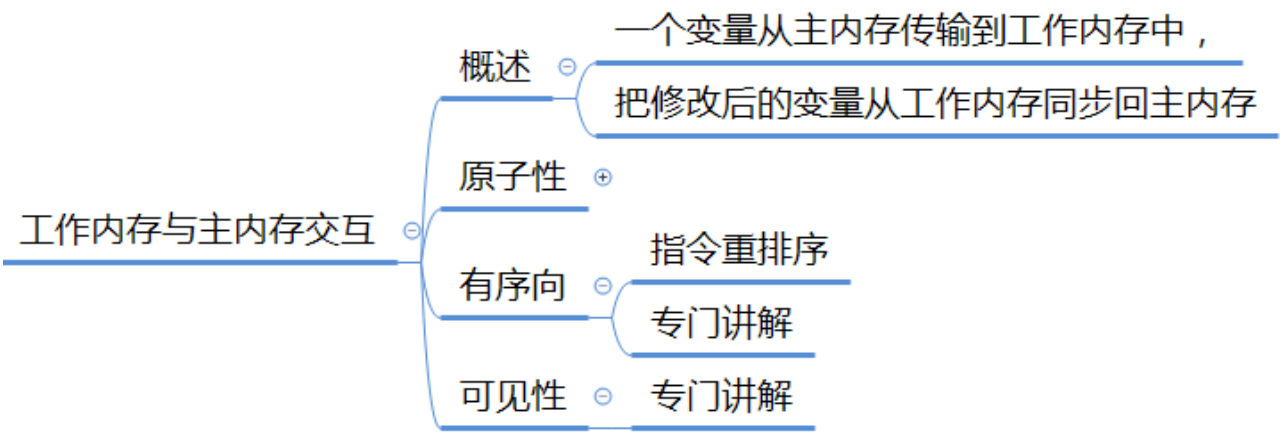


* 工作内存与主内存交互

  * 概述

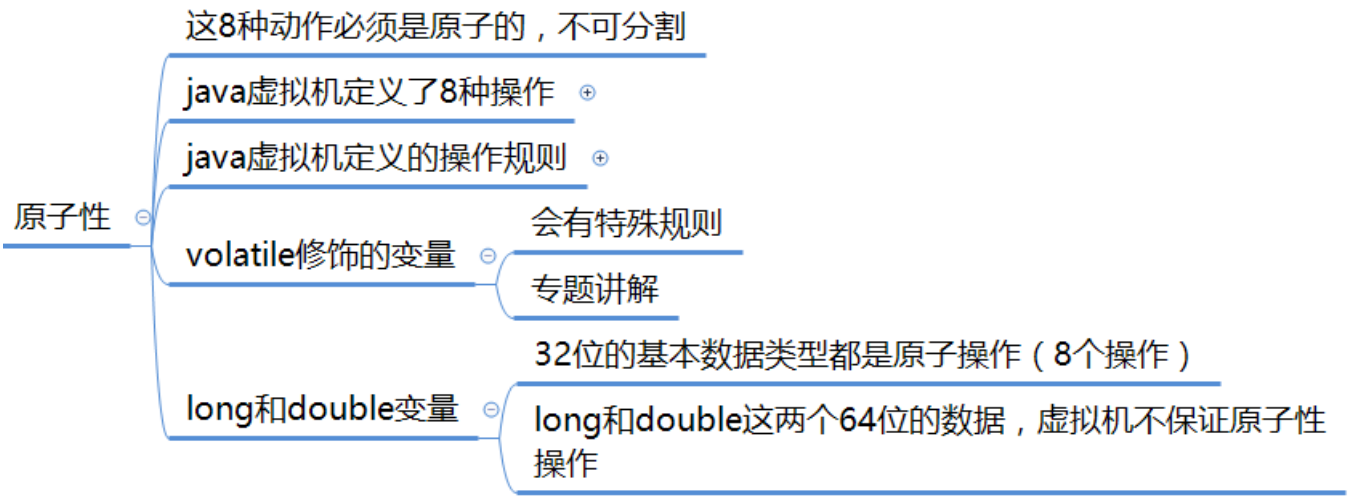


* 交互



* 原子性

* 概述


原子性
这8种动作必须是原子的，不可分割
java虚拟机定义了8种操作
java虚拟机定义的操作规则
volatile修饰的变量
会有特殊规则
专题讲解
long和double变量
32位的基本数据类型都是原子操作（8个操作）
long和double这两个64位的数据，虚拟机不保证原子性操作

* 操作

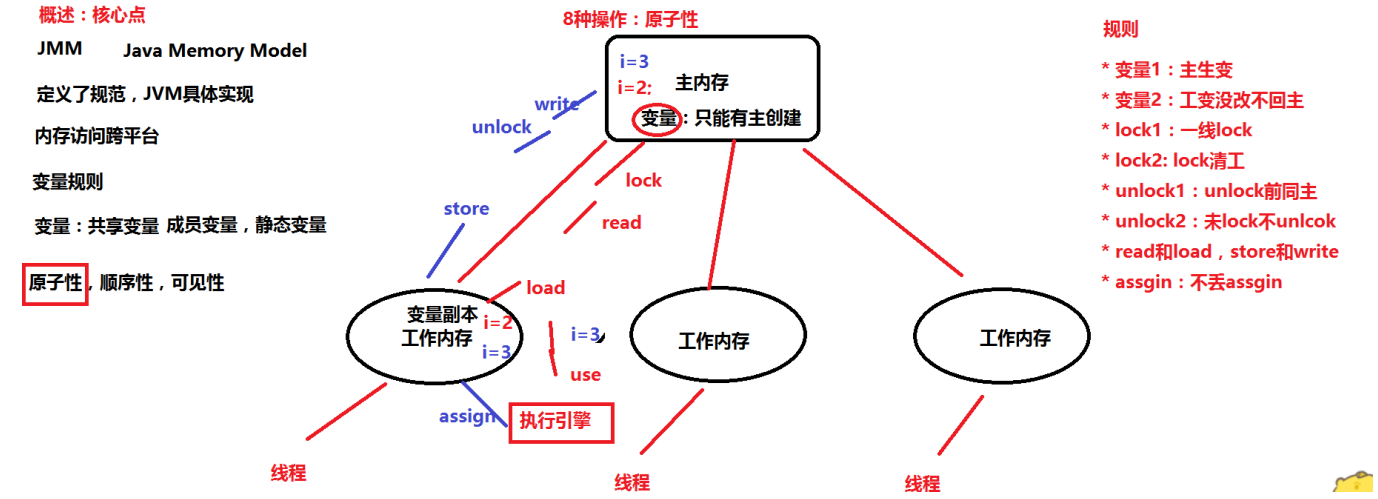

java虚拟机定义了8种操作
lock(锁定)
作用于主内存的变量，一个变量在同一时间只能一个线程锁定，该操作表示这条线成独占这个变量
unlock(解锁)
作用于主内存的变量，表示这个变量的状态由处于锁定状态被释放，这样其他线程才能对该变量进行锁定
read(读取)
作用于主内存变量，表示把一个主内存变量的值传输到线程的工作内存
load(载入)
作用于线程的工作内存的变量，表示把read操作从主内存中读取的变量的值放到工作内存的变量副本中
use(使用)
作用于工作内存，把工作内存的变量值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作
assign(赋值)
作用于工作内存，把执行引擎收到的值赋给工作内存的变量，虚拟机遇到赋值字节码时候执行这个操作
store(存储)
作用于工作内存，把变量的值传输到主内存中
write(写入)
作用于主内存的变量，把store操作从工作内存中得到的变量的值放入主内存的变量中

* 规则

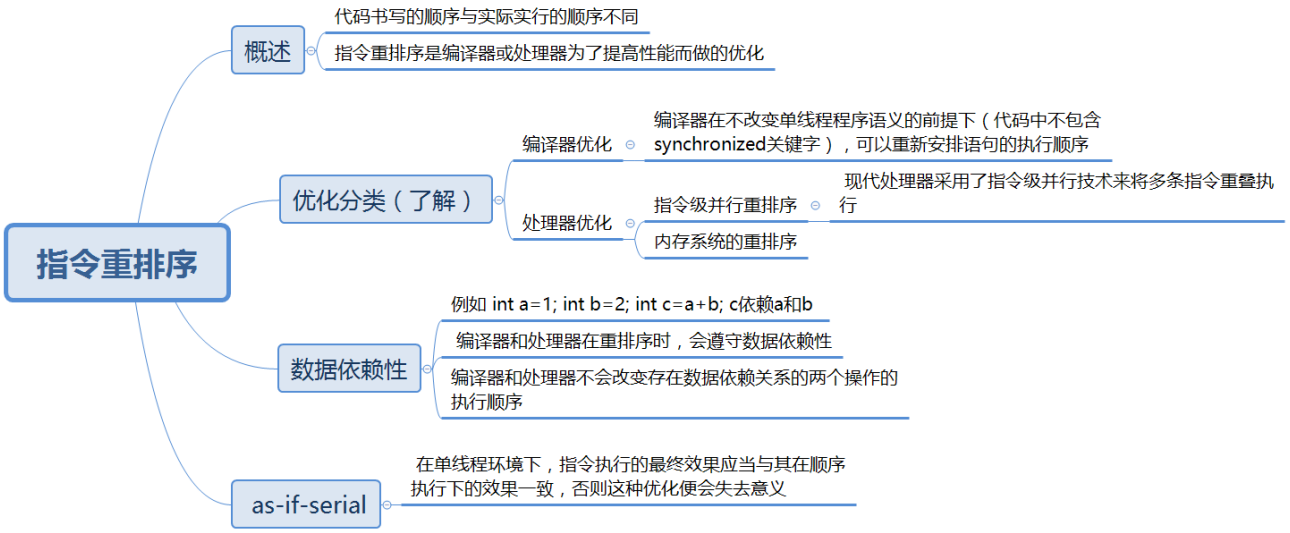
java虚拟机定义的操作规则

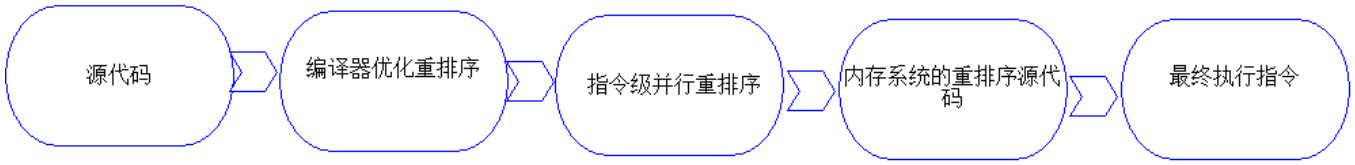
- 变量只能在主内存中产生
- 不允许一个线程回写没有修改的变量到主内存
- 一个变量同一时刻只允许一条线程进行lock操作
- 对一个变量进行lock操作，那么将会清空工作内存中此变量的值
- 对一个变量执行unlock之前，必须先把变量同步回主内存中
- 不允许对未lock的变量进行unlock操作
- 不允许read和load,store和write操作之一单独出现
- 不允许一个线程丢弃它最近的assign操作

* 画图讲解



* 能够理解指令重排序






```
* 案例一
 * 在单线程的情况
public static void main(String[] args) {
        int a=1;  // 1
        int b=2;  // 2
        int c=a+b;  // 3
}
结果：
* 按程序顺序执行结果：1 - 2 - 3 ---> c=3
* 重排序后的执行结果：2 - 1 - 3 ---> c=3

* 温馨提醒：
 * 数据依赖性：例如c依赖a和b
  * 编译器和处理器在重排序时，会遵守数据依赖性，
  * 编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序
 * as-if-serial:
  * 在单线程环境下，指令执行的最终效果应当与其在顺序执行下的效果一致，
    否则这种优化便会失去意义

* 案例二
```

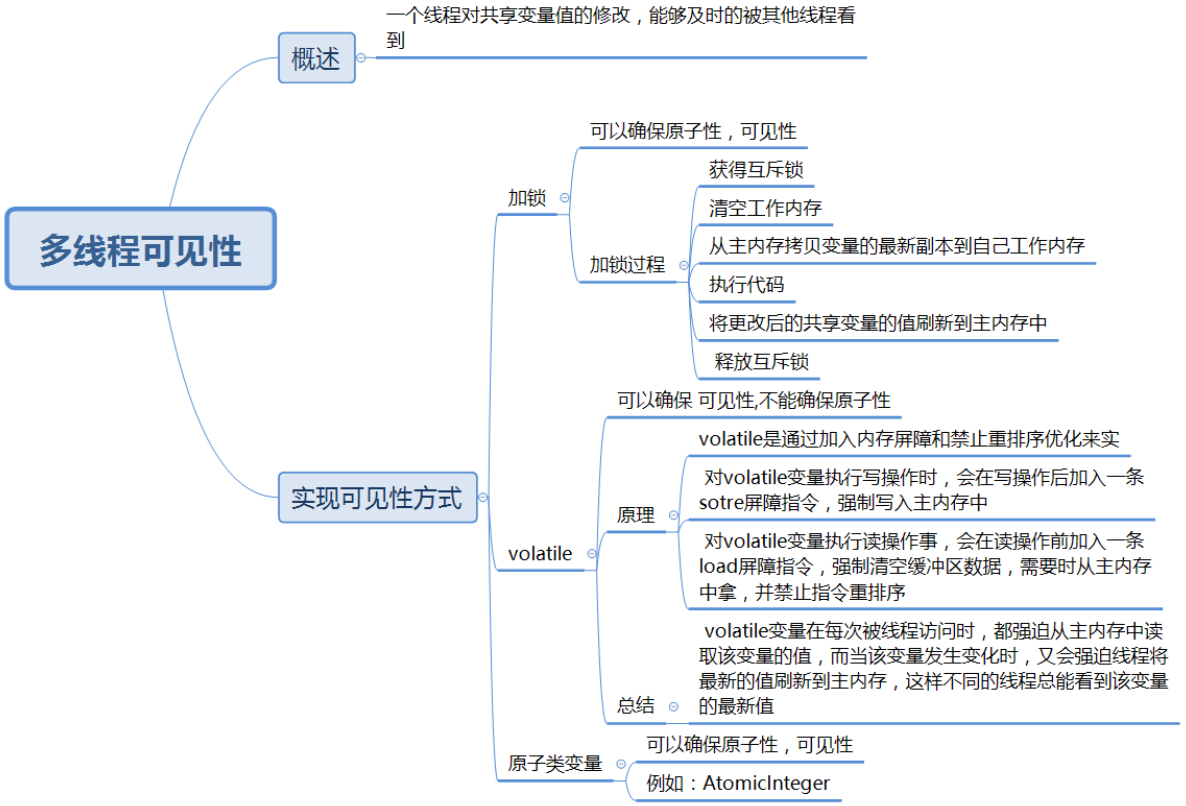
```
 * 在多线程情况
public class Main {
    public static int a;
    public static boolean flag;
    public static void main(String[] args) throws InterruptedException {
        while (true){
           Thread t1= new Thread(new Runnable() {
                @Override
                public void run() {
                    a=1; // 1
                    flag=true; // 2
                }
            });
         Thread t2= new Thread(new Runnable() {
             @Override
             public void run() {
                 if (flag){ // 3
                     if(a==0){ // 4
                         System.err.println("bingo");
                     }
                 }
             }
         });
         t1.start();
         t2.start();
         t1.join();
         t2.join();
         a=0;
         flag=false;
        }

    }
}

 * 按程序顺序执行结果: 1 - 2 -3
 * 重排序后的执行结果: 2  - 3 - 4
* 解决方案
  * 1 禁止指令重排序
    public volatile static int a;
    public volatile static boolean flag;
```

```
61    * 2 加锁（性能不好）
```

* 结果：等待一段时间

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
bingo
bingo
bingo
```

* 能够理解多线程的内存可见性



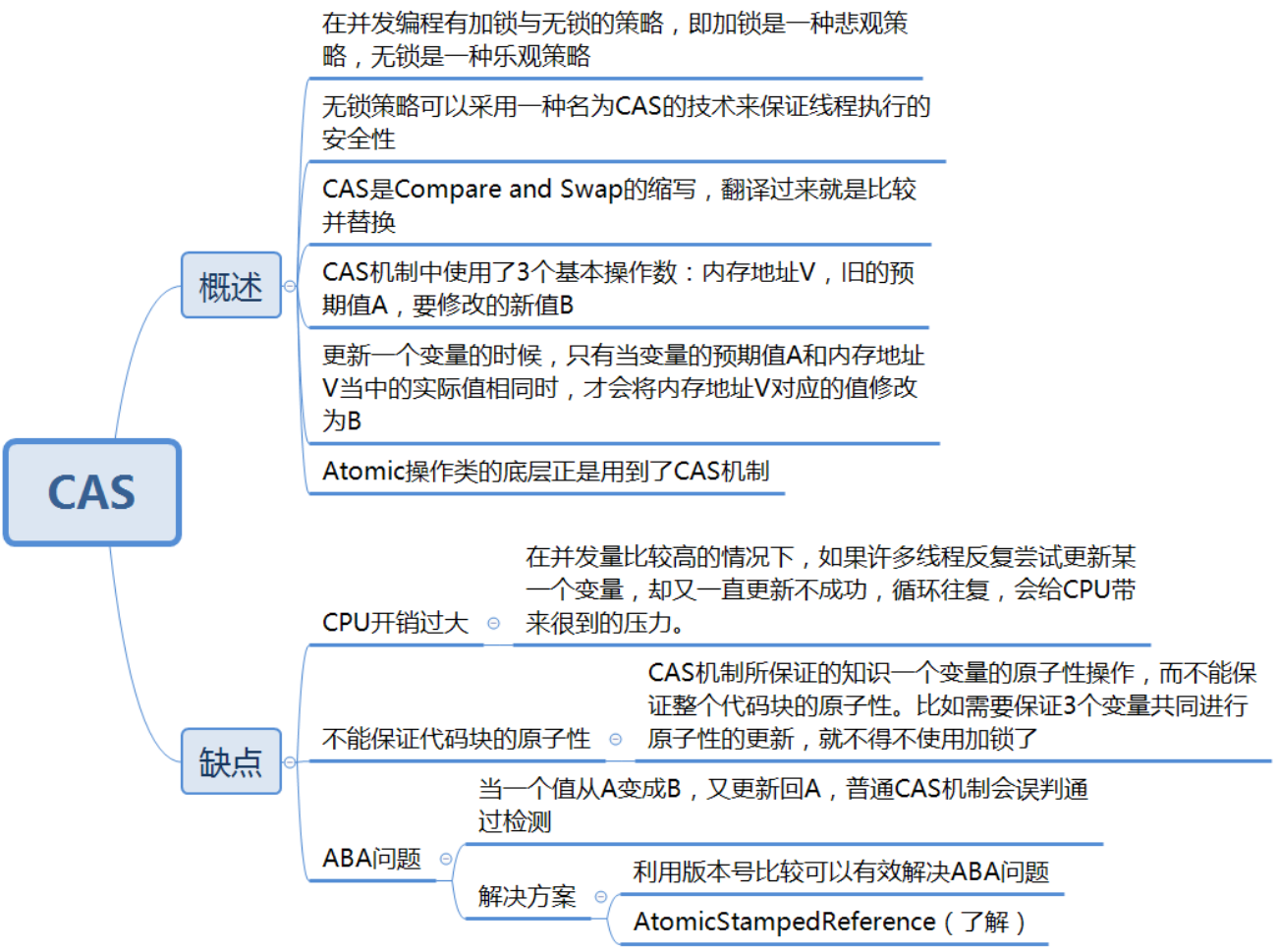

```
1  案例
2  public class Main {
3      public static void main(String[] args)  {
4        Game game=new Game();
5        game.start();
6          try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        game.command=-1;
      System.out.println("main over...");
    }
   static class Game extends Thread{
       public int command;
       @Override
       public void run() {
           System.out.println("游戏开始...");
           while (true){
               if(command==-1){
                   break;
               }
           }
           System.out.println("游戏结束");
       }
   }
}

* 结果
游戏开始...
main over...
* 发现：在main线程修改变量，无法更新到game线程，线程不可见性
* 解决方案
  * volatile
     * public volatile int command;
  * 加锁
public class Main3 {
    public static void main(String[] args)  {
      Game game=new Game();
      game.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        game.setCommand(-1);
```

```
      System.out.println("main over...");
    }
   static class Game extends Thread{
       private int command;
       public synchronized void setCommand(int command){
           this.command=command;
       }
       public synchronized int getCommand(){
           return command;
       }
       @Override
       public void run() {
           System.out.println("游戏开始...");
           while (true){
               if(getCommand()==-1){
                   break;
               }
           }
           System.out.println("游戏结束");
       }
   }
}
    * 原子性变量
public class Main4 {
    public static void main(String[] args)  {
      Game game=new Game();
      game.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        game.command.set(-1);
      System.out.println("main over...");
    }
   static class Game extends Thread{
       public AtomicInteger command;
       public Game(){
           command=new AtomicInteger(0);
       }
```

```
87          @Override
88          public void run() {
89              System.out.println("游戏开始...");
90              while (true){
91                  if(command.get()==-1){
92                      break;
93                  }
94              }
95              System.out.println("游戏结束");
96          }
97      }
98  }
99
```

* 能够理解线程安全性CAS



* 画图解释CAS机制

银行转钱案例

A1 : 100 -- 转 20

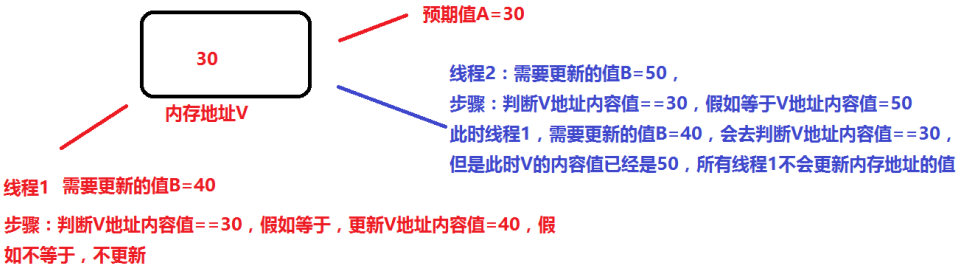
V=100,A=100,B=80

V=70

A2 : 100 -- 转30

V=100,A=100,B=70

V=70

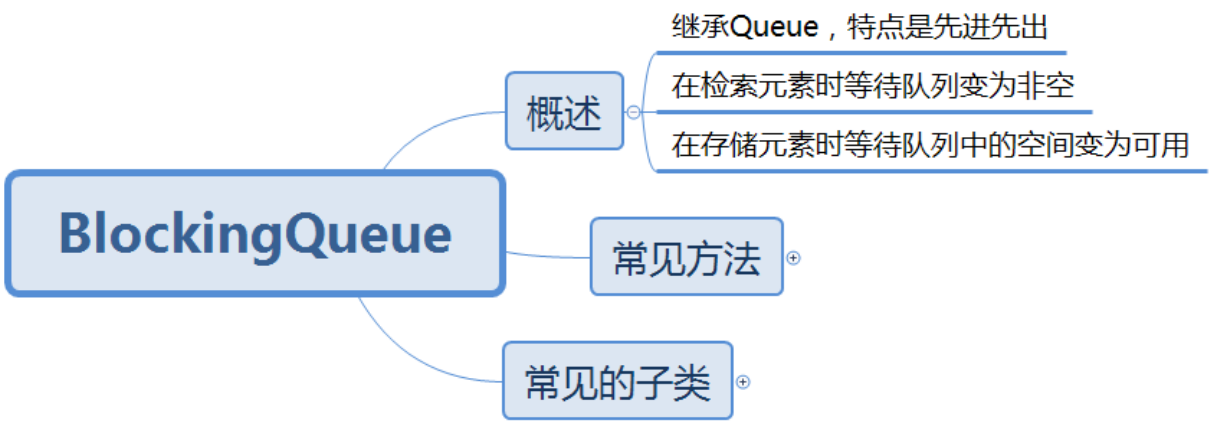



```
* 案例
  * num++ 不是原子性操作
    num=num+1
package com.lg.test7;

/**
 * @author xiaozhao
 */
public class Main6 {
    public static int num;
    public static void main(String[] args)  {
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    add();
                }
            }).start();
        }
    }
    public static  void add(){
        num++;
        System.out.println("num = " + num);
    }
}
* 结果：可能会出现重复数字

* 解决方案：原子性
```

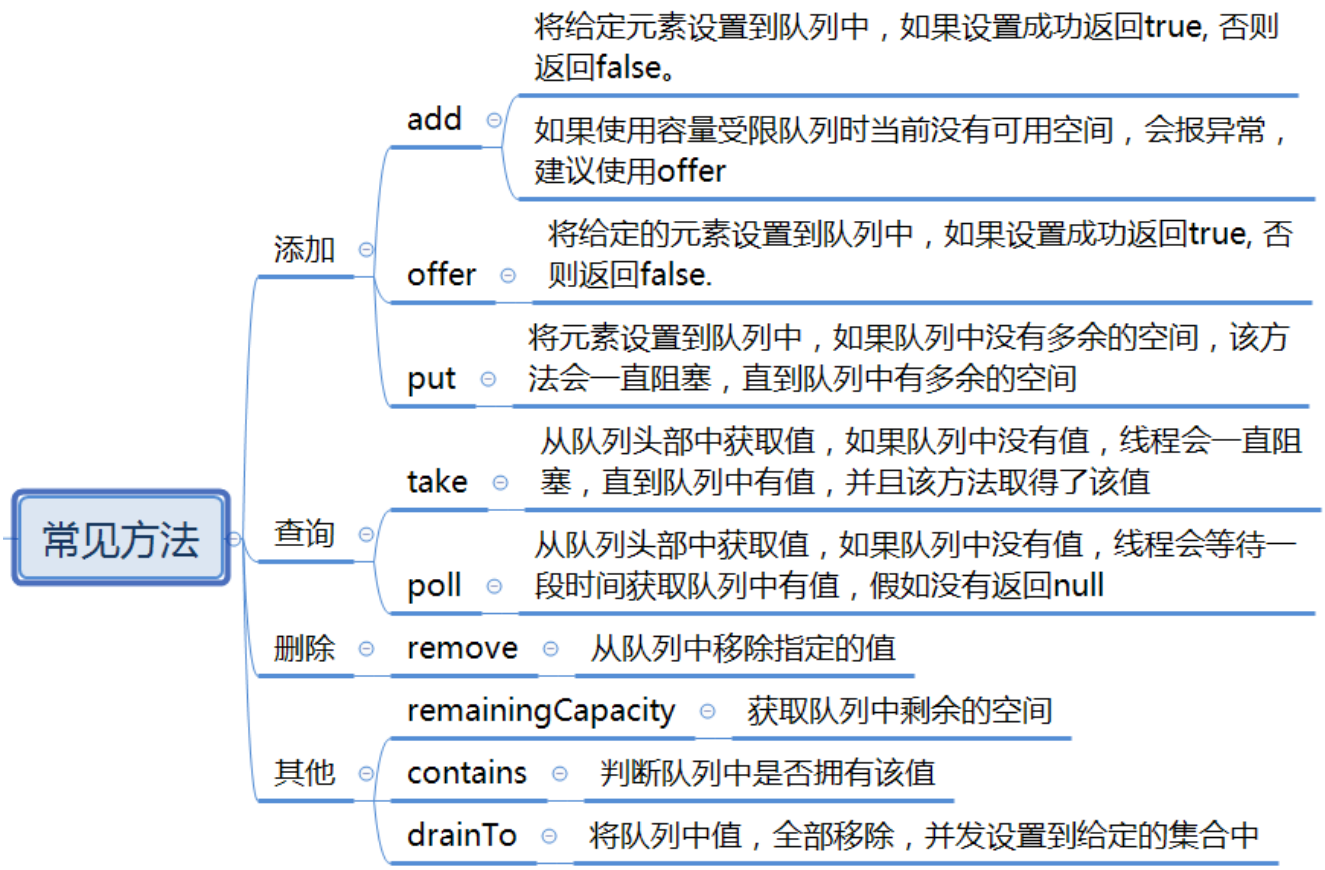
```
 * 加锁
 * 用原子类
public class Main {
    public static AtomicInteger num=new AtomicInteger(0);
    public static void main(String[] args)  {
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    add();
                }
            }).start();
        }
    }
    public static  void add(){
        int value=num.incrementAndGet();
        System.out.println("value = " + value);
    }
}

```

* 能够掌握BlockingQueue相关阻塞队列

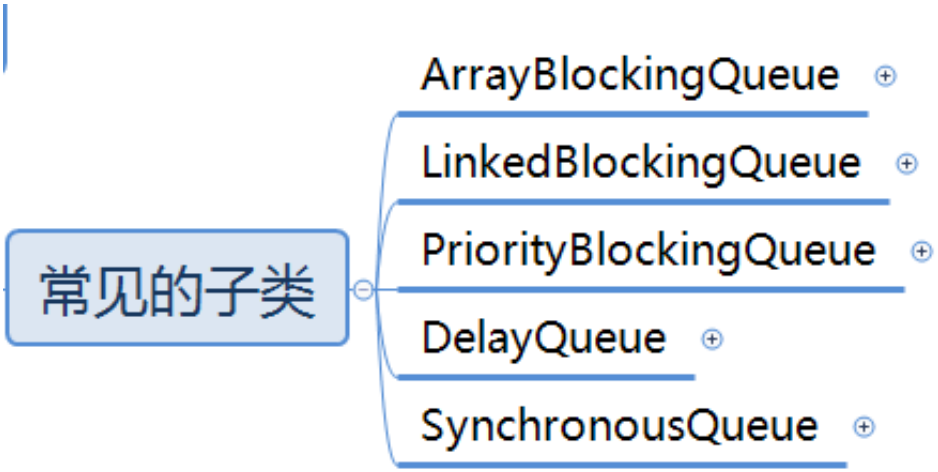
 * 概述



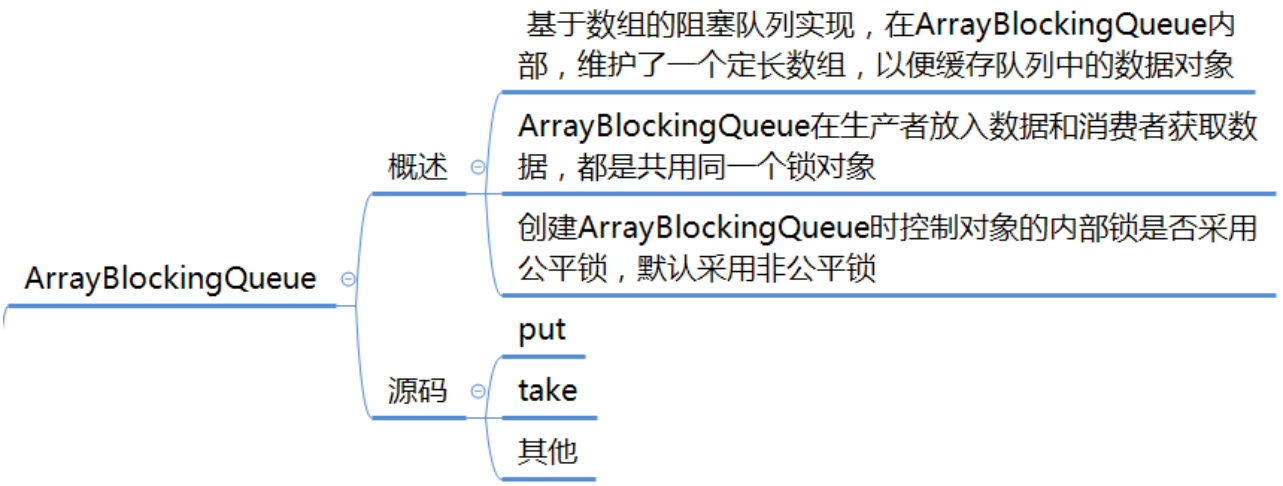
 * 常见方法



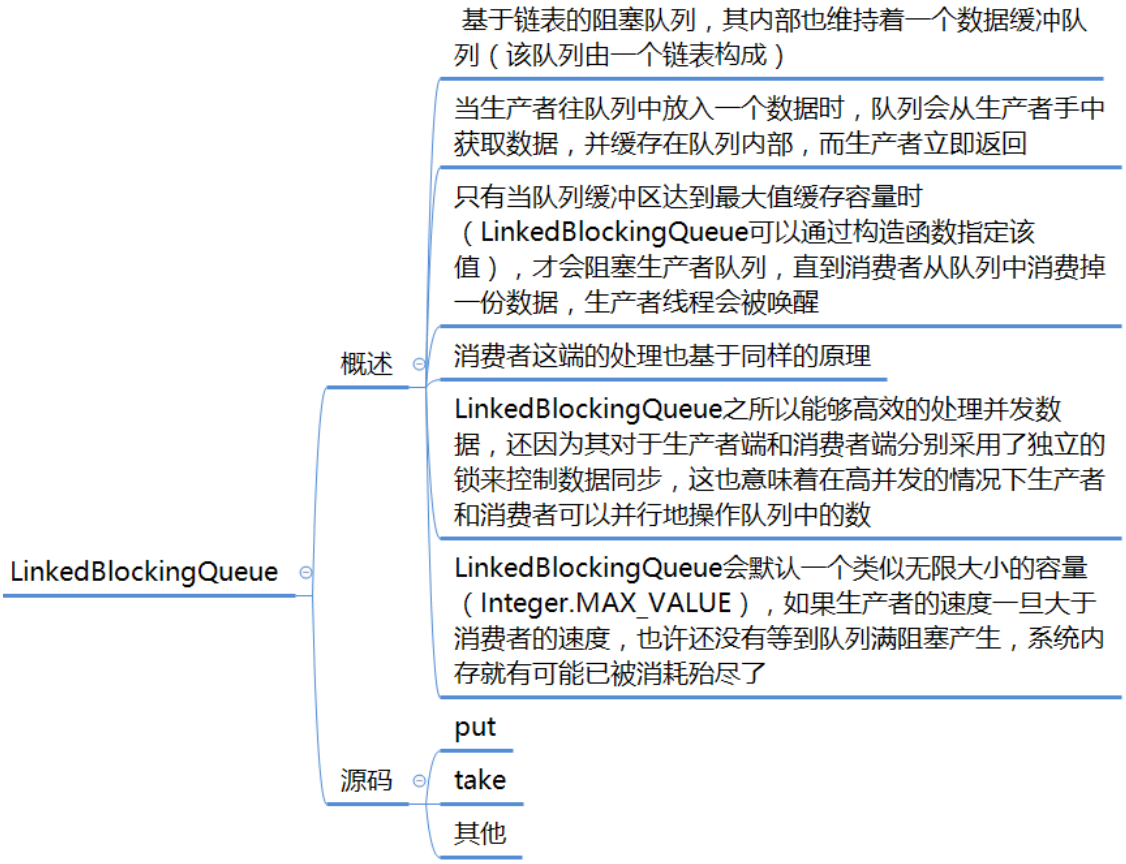
* 常见的子类



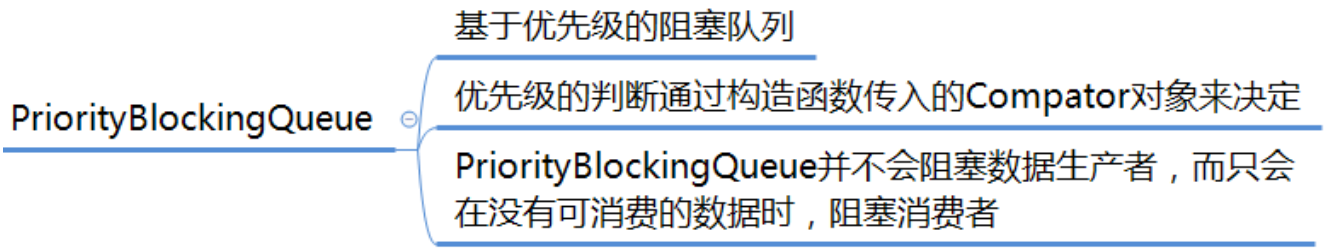
* ArrayBlockingQueue



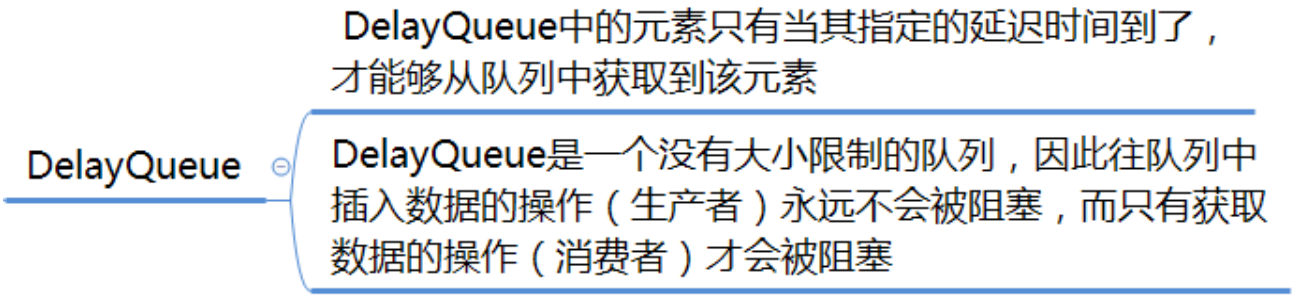
* LinkedBlockingQueue



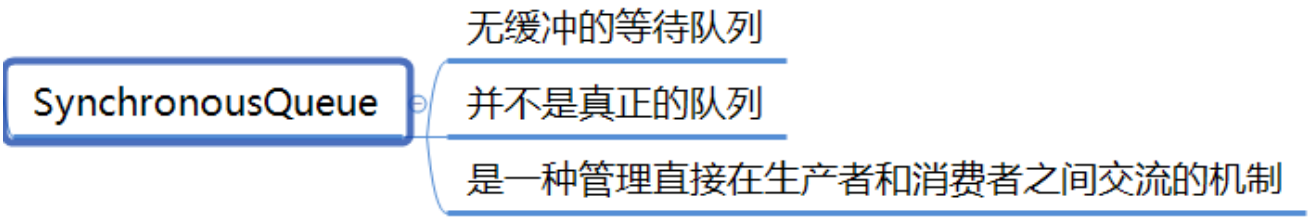
* PriorityBlockingQueue



* DelayQueue



* SynchronousQueue




```
* 案例：ArrayBlockingQueue
 * 生产者
public class Producer1 implements Runnable {
    private BlockingQueue mQueue;
    public Producer1(BlockingQueue queue){
        this.mQueue=queue;
    }
    @Override
    public void run() {
        // 一直在生产
        Random r=new Random();
        while (true){
                // 没有数据，就生产
            while (mQueue.size()<=0) {
                try {
                    Thread.sleep(2000);
                    int num = r.nextInt(50);
                    mQueue.put(num);
                    System.out.println(Thread.currentThread().getName() + "线程
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }
}
  * 消费者
public class Consumer1 implements Runnable {
    private BlockingQueue mQueue;
    public Consumer1(BlockingQueue queue){
       this.mQueue=queue;
    }
    @Override
    public void run() {
        while (true){
                while (mQueue.size() > 0) {
                    try {
                        Thread.sleep(1000);
                        System.err.println(Thread.currentThread().getName()+"线
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
        }
    }
}
 * Main
public class Main {
    public static void main(String[] args)  {
        BlockingQueue blockingQueue=new ArrayBlockingQueue(16);
        for (int i = 0; i < 5; i++) {
            // 生产者
            new Thread(new Producer1(blockingQueue),"生产者"+i){}.start();
            // 消费者
            new Thread(new Consumer1(blockingQueue),"消费者"+i){}.start();
        }
    }

}

* 案例二：LinkedBlockedQueue
  * ArrayBlockingQueue 替换成LinkedBlockedQueue

* 案例三：PriorityBlockingQueue
```

```java
@Data
@AllArgsConstructor
public class Task implements Comparable<Task> {
    private int id;
    private String name;
    @Override
    public int compareTo(Task o) {
        return this.getId()-o.id ;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Task> queue=new PriorityBlockingQueue<Task>();
        Task task1=new Task(1,"xiaohei");
        Task task3=new Task(3,"xiaobai");
        Task task2=new Task(2,"xiaoming");
        queue.put(task1);
        queue.put(task2);
        queue.put(task3);

        for (Task task : queue) {
            System.out.println(task);
        }
        List list=new ArrayList();
        queue.drainTo(list);
        queue.take();
        System.out.println("main over");
    }
}

* 案例网吧上网 :DelayQueue
 * 实现 Delayed接口
@Data
public class Customer implements Delayed {
    private int id;
    private String name;
    private long endTime;

    @Override
```

```
    public long getDelay(TimeUnit unit) {
        return this.endTime-System.currentTimeMillis();
    }

    public Customer(int id, String name, long endTime) {
        this.id = id;
        this.name = name;
        // 加上当前时间
        this.endTime = endTime+System.currentTimeMillis();
    }

    @Override
    public int compareTo(Delayed o) {
        return (int) (this.endTime-((Customer) o).endTime);
    }
}
 * DownTask
public class DownTask implements Runnable {
    private BlockingQueue<Customer> mQueue;
    public DownTask(BlockingQueue<Customer> blockingQueue){
            this.mQueue=blockingQueue;
    }
    @Override
    public void run() {
        while (true) {
            try {
                Customer customer = mQueue.take();
                System.out.println("编号: " + customer.getId() + " 姓名: " + cus
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

* Main
public class Main {
    public static void main(String[] args) {
        BlockingQueue<Customer> queue=new DelayQueue<Customer>();
        Customer c1=new Customer(1001,"刘备",1000*5);
```

```
        Customer c2=new Customer(1002,"关羽",1000*10);
        Customer c3=new Customer(1003,"张飞",1000*20);
        queue.add(c1);
        queue.add(c2);
        queue.add(c3);
        new Thread(new DownTask(queue)).start();
        SimpleDateFormat sdf =new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
        while (true){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(sdf.format(new Date()));
        }
    }
}

* 案例：SynchronousQueue
  * 生产者
public class Producer1 implements Runnable {
    private BlockingQueue mQueue;
    public Producer1(BlockingQueue queue){
        this.mQueue=queue;
    }
    @Override
    public void run() {
        // 一直在生产
        Random r=new Random();
        while (true){
                try {
                    Thread.sleep(2000);
                    int num = r.nextInt(50);
                    mQueue.put(num);
                    System.out.println(Thread.currentThread().getName() + "线程
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

        }
```

```
185     }
186 }
187 
188   * 消费者
189 public class Consumer1 implements Runnable {
190     private BlockingQueue mQueue;
191     public Consumer1(BlockingQueue queue){
192        this.mQueue=queue;
193     }
194     @Override
195     public void run() {
196         while (true){
197                     try {
198                         Thread.sleep(1000);
199                         System.err.println(Thread.currentThread().getName()+"线
200                     } catch (InterruptedException e) {
201                         e.printStackTrace();
202                     }
203         }
204     }
205 }
206 
207  * Main
208 public class Main {
209     public static void main(String[] args)  {
210         BlockingQueue blockingQueue=new SynchronousQueue();
211             // 生产者
212         new Thread(new Producer1(blockingQueue),"生产者"){}.start();
213             // 消费者
214         new Thread(new Consumer1(blockingQueue),"消费者"){}.start();
215     }
216 }
217 
```