hashMap为何线程不安全

HashMap在put的时候，插入的元素超过了容量（initialCapacity * loadFactor）的范围就会触发扩容操作，就是rehash，
这个会重新将原数组的内容重新hash到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行put操作，如果hash值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。
线程1：rehash后，第一个链表顺序为： 0 -> 4 挂起
线程2：rehash后，第一个链表顺序为： 4 -> 0

再次执行线程1：存储下来的链表就是 0 -> 4 -> 0。即node.next指向了之前的元素，形成死循环。CPU飙升！！

二、线程间通信

线程之间的通信主要是采用wait，notify，notifyall方法

也可以通过CountDownLatch实现通信

典型案例：

take中的notify唤醒put中的wait方法。

```
public class EventQueue {
    @Data
    @AllArgsConstructor
    static class Event {
        private int num;
    }

    private int max;
    private final LinkedList<Event> eventQueue = new LinkedList();

    private final static int DEFAULT_MAX_EVNET_SIZE = 10;

    public EventQueue() {
        this(DEFAULT_MAX_EVNET_SIZE);
    }

    public EventQueue(int max) {
        this.max = max;
    }

    public void put(Event event) {
        synchronized (eventQueue) {
            try {
                if (eventQueue.size() >= max) {
                    System.out.println("i am full, please wait");
                    // 队列满了，put等待take消费掉一条消息后，再被唤醒
                    eventQueue.wait();
                }
            } catch (InterruptedException ex) {
                System.out.println("i am interrupted");
            }
            eventQueue.addLast(event);
            // 唤醒take里面的eventQueue.wait()
            eventQueue.notify();
        }
    }

    public Event take() {
        synchronized (eventQueue) {
            try {
                if (eventQueue.isEmpty()) {
                    System.out.println("i am empty,can not take event, please wait");
                    eventQueue.wait();
                }
            } catch (InterruptedException ex) {
                System.out.println("i am interrupted");
            }
            Event pop = eventQueue.removeFirst();
            System.out.println(pop);
            // 唤醒put里面的eventQueue.wait()
            eventQueue.notify();
            return pop;

        }
```

```
    }
}
```

在使用了锁的class文件中(字节码文件)，会看一个monitorenter 指令以及多个 monitorexit 指令。表示锁的进去和退出。

1.线程休息室 (wait set)

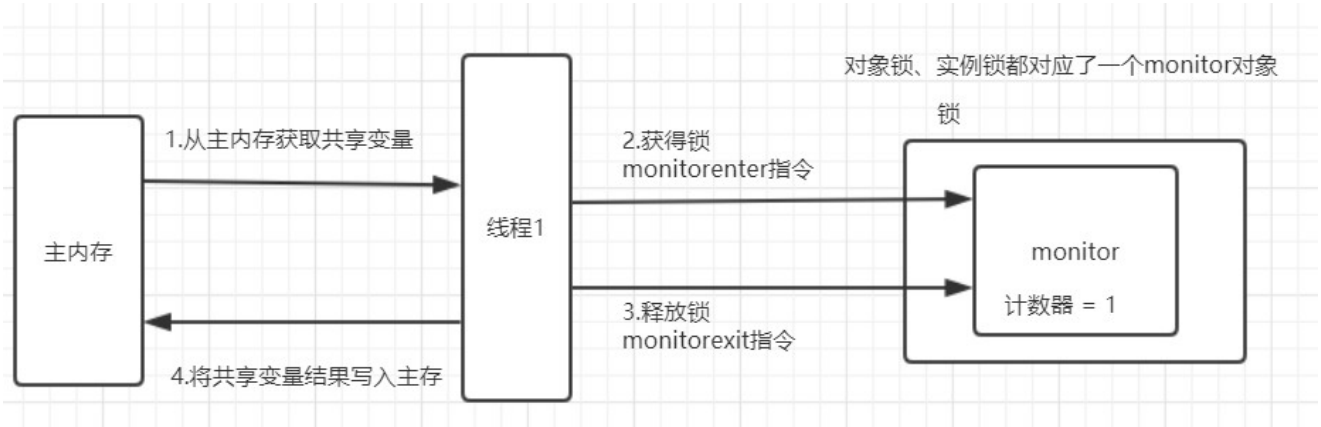
当Thread调用锁对象的wait方法后，该Thread就会被加入该对象monitor关联的`wait set`中，并释放monitor的所有权

- 锁对象调用notify方法：其中一个线程从wait set弹出（顺序不确定）
- 锁对象调用notifyAll方法：`wait set`所有线程都被弹出

2.synchronized的缺点

synchronized关键字包含两个JVM指令：monitor enter 和monitor exit。

在monitor exit运行成功后，共享变量的值必须刷入主内存中，因此monitor enter成功之前都必须从主内存中获取数据，而不是从缓存中。



synchronized只能用来修饰方法和代码块，不能修饰类的变量。

缺点：

- 无法控制阻塞时长。无法设置超时的功能
- 一旦阻塞，便不可中断

```
public class SyncTest {
    public static void main(String[] args) throws InterruptedException {
        SyncTest syncTest = new SyncTest();
        Thread t1 = new Thread(syncTest::syncMethod, "t1");
        t1.start();
        t1.interrupt();   // t1持有monitor的锁。调用interrupt方法可以直接中断

        TimeUnit.MILLISECONDS.sleep(10);  // 确保t1先进入syncMethod

        Thread t2 = new Thread(syncTest::syncMethod, "t2");
        t2.start();
        t2.interrupt();   // t2为等待t1结束进入了阻塞状态。调用interrupt方法无效
    }

    public synchronized void syncMethod() {
        try {
            System.out.println("sleep 1 hour");
            TimeUnit.HOURS.sleep(1);
        } catch (InterruptedException ex) {
            System.out.println("被中断");
        }
    }
}

说明：
  1.T2线程何时获得执行syncMethod方法，完全取决于T1何时释放。无法实现T2最多等1分钟就放弃的功能
  2.一旦T2进入阻塞状态，他将无法被中断。因为synchronized阻塞无法向sleep和wait一样，捕捉到interrupt信号。所以现在争抢锁的过程中，无法获得锁的线程只能一直等待下去
```

monitor enter

每个对象或实例都与一个monitor对象相关联，monitor的lock锁同一时间只会被一个线程获得。

- 如果monitor的计数为0，没有线程获得该monitor的锁
- 如果线程重入，monitor的计数+1
- 其他线程想获得monitor的所有权，会陷入阻塞状态直到monitor的计数为0，才能尝试获得。

3.hook线程

在JVM退出时，Hook线程会启动执行。

开发中，比如防止某个程序被重复启动。在启动时，创建一个lock文件，中断时会删除这个lock文件。

在mysql服务器、zookeeper、kafka等系统都能看到lock文件的存在。

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    System.out.println("删除hook文件");
}));
```

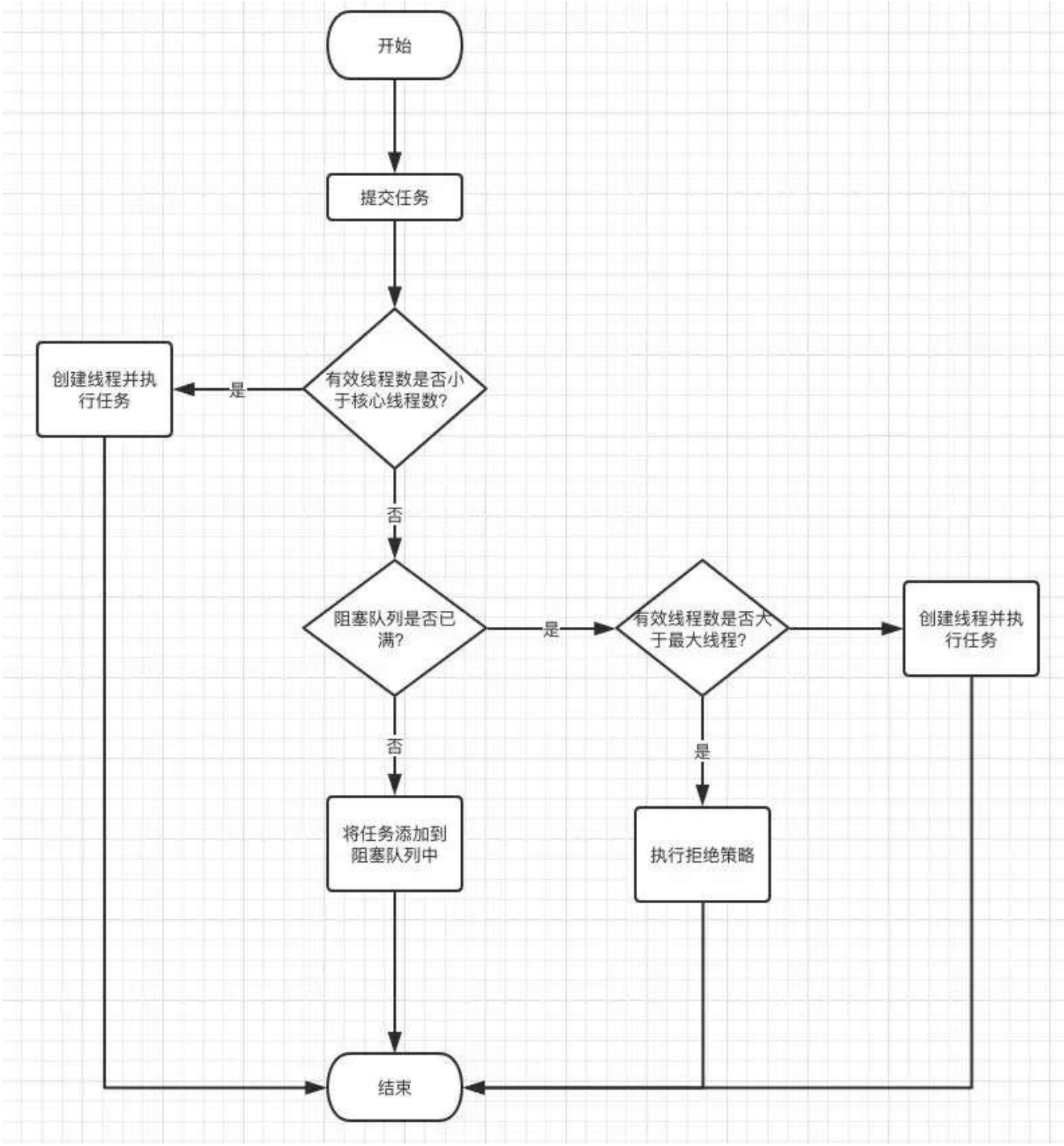
注意：

- hook只有在正常退出才会执行。kill -9时，lock文件不会被清除
- hook中也可以执行释放资源的操作。比如关闭socket链接、数据库连接

- 尽量不要在hook中进行一些耗时很长的操作。会导致程序迟迟不能退出。

4.线程池的原理

```
Q. 线程池是什么时候创建线程的？
A. 任务提交的时候

Q.线程池如何执行任务
（1）当线程数小于核心线程数时，创建线程。
（2）当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
（3）当线程数大于等于核心线程数，且任务队列已满
    1）若线程数小于最大线程数，创建线程
    2）若线程数等于最大线程数，执行拒绝策略。

Q. 什么时候会触发reject策略？
A.队列满并且maxthread也满了， 还有新任务，默认策略是reject

Q. core到maxThread之间的线程什么时候会die?
A.  没有任务时，或者抛异常时。
   core线程也会die的，core到maxThread之间的线程有可能会晋升到core线程区间，
   core max只是个计数，线程并不是创建后就固定在一个区间了

Q. task抛出异常，线程池中这个work thread还能运行其他任务吗?
A. 不能。但是会创建新的线程，新线程可以运行其他task。有异常时旧的thread会被删除（GC回收），再创建新的thread， 即有异常时，旧thread不可能再执行新的任务。
```



4.1 java的四种线程池

Java通过Executors提供四种线程池，分别为：
1,newCachedThreadPool
创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
2,newFixedThreadPool
创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3,newScheduledThreadPool
创建一个定长线程池，支持定时及周期性任务执行。
4,newSingleThreadExecutor
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

4.2 拒绝策略

可以理解为达到最大线程数时，一个回调函数。

JDK默认有4个实现类，也可以自定义拒绝策略。

`AbortPolicy（默认）是抛出异常`，DiscardPolicy是直接将不能执行的任务删除。

```
从字面上理解为拒绝处理者,可以理解为当任务数大于maxmumPoolSize后的一个回调方法,RejectedExecutionHandler
本身是一个接口,jdk本身对他有4个实现类

// 线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度
CallerRunsPolicy

// 处理程序遭到拒绝将抛出运行时RejectedExecutionException;
AbortPolicy(默认)

// 不能执行的任务将被删除
DiscardPolicy

// 如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）
DiscardOldestPolicy
```

建议：

自定义一个reject策略，如果线程池无法执行更多的任务了，此时建议可以把task信息持久化写入磁盘中，后续等待线程池的工作负载降低了。后台再启动一个线程，从磁盘读取task，从新提交到线程池中去执行。

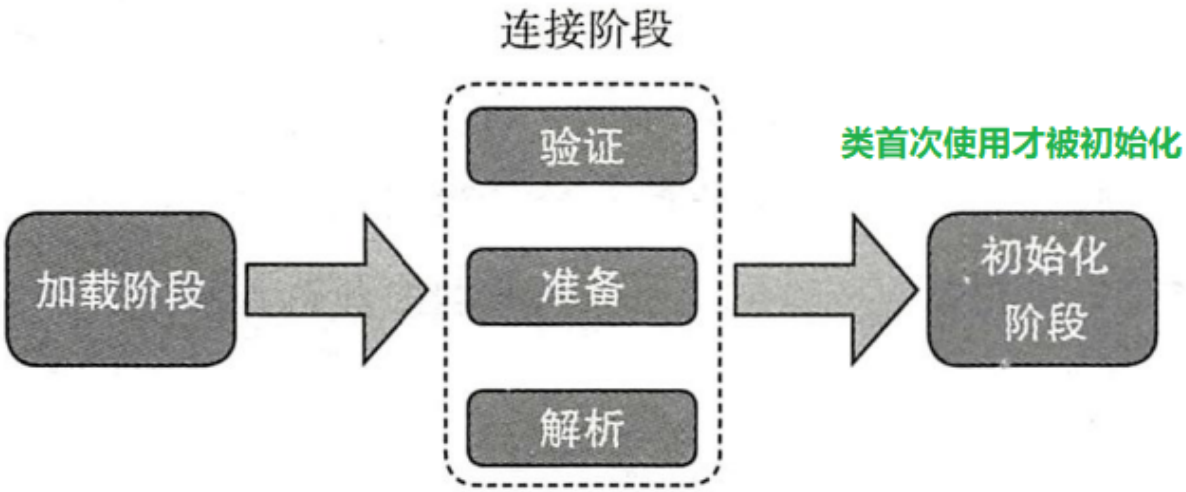
4.3 使用线程池的注意事项

```
newFixedThreadPool => return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
阻塞队列基本是无界的。不会触发线程池的拒绝策略。
可能队列里面积累的任务数目，直接将内存撑爆了。。。

------------------------------------------------------------------------
newCachedThreadPool => return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
最大线程数为Integer.MAX_VALUE，而且阻塞队列是无界的，不会触发线程池的拒绝策略。
可能会创建很多线程池，造成浪费。而且单机线程数不能无限上升，在高并发情况下，如果线程调用的外部服务异常了，
每个都在等待，会导致新来的请求又重新创建线程。结果就有几千个线程。。。机器挂了
```

三、类加载过程

```
类被加载后，其class对象在堆内存中，class对象的数据结构在方法区中
```

1.类加载过程阶段



- 加载阶段

  查找和加载类的二进制文件。其实就是.class文件

- 连接阶段
  - 验证：确保类文件的正确性，不会出现危害JVM自身安全的代码
  - 准备：为静态变量分配内存，并为其初始化默认值
  - 解析：将类中的符号引用转换成直接引用
- 初始化阶段

  为类的静态变量赋值（程序指定的值）

> 程序启动后，不是每个类都会被初始化。
> JVM对类的初始化有延迟机制--lazy方式，当一个类被第一次使用才会被初始化。并且一个Class只会被初始化一次

2.类的连接阶段

类的连接阶段被分为三个阶段

2.1 验证

验证的目的是确保class文件的字节流所包含的内容：符合JVM规范的要求；不糊出现危害JVM自身安全的代码

- 验证文件格式：　主要验证class文件的文件类型、JDK版本、MD5值、变量/常量的类型是否正确
- 元数据验证：　　验证是否符合JVM规范。就是代码对不对。比如final修饰的类不允许被继承，重载的方法是否合法、
- 字节码验证：　　（比较复杂）验证控制流程。比如循环

2.2 准备

静态变量的内存被分配到方法区（metaspace）

实例变量的内存被分配到堆内存

3.类的主动使用

JVM虚拟机规范规定：

每个类或者接口只有在`首次主动使用`才会对其进行初始化。

(现在在运行期间也会进行预判并初始化某个类)

JVM规定了以下6中类的主动使用场景：

```
public class test {
    public static void main(String[] args) {
        try {
          // 1.new关键字
        TestInitFather testInitFather = new TestInitFather();
          // 2.访问类的静态变量
        int i = TestInitFather.i;
          // 3.访问类的静态方法
        TestInitFather.print();
          // 4.对某个类进行反射
        Class<?> clazz = Class.forName("com.wm.concurrent.chapter9.TestInitFather");
          // 5.初始化子类会导致父类的初始化
        TestInitSon testInitSon = new TestInitSon();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    // 6.执行main函数所在的启动类会被初始化
    static {
        System.out.println("main函数所在的启动类：初始化");
    }
}
----------------------------------
public class TestInitFather {
    static {
        System.out.println("TestInitFather初始化");
    }
    public static int i =10;
    public static void print(){
        System.out.println("TestInitFather的静态方法");
    }
    // 访问静态常量不会导致类的初始化
    public final static int MAX =10;
}
----------------------------------
public class TestInitSon extends TestInitFather {
    static {
        System.out.println("testInitSon初始化");
    }
}
```

4.类的被动使用

除了以上六种类的主动使用，其余都是被动使用，不会导致类的加载和初始化

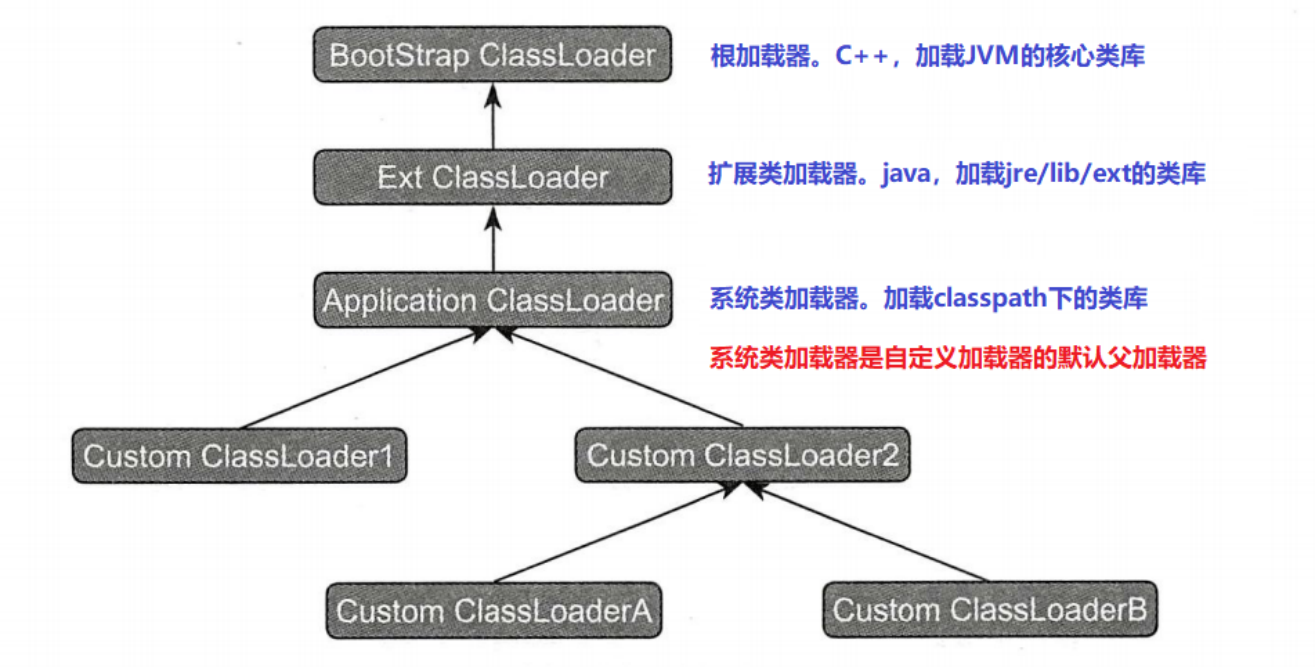
- 构造某个类的数组时不会初始化
- 引用类的静态常量（ final static修饰）

  在编译阶段javac会对MAX的值生成一个ConstantValue属性，并直接赋值10

```
TestInitFather[] arr = new TestInitFather[10]
TestInitFather.MAX
```

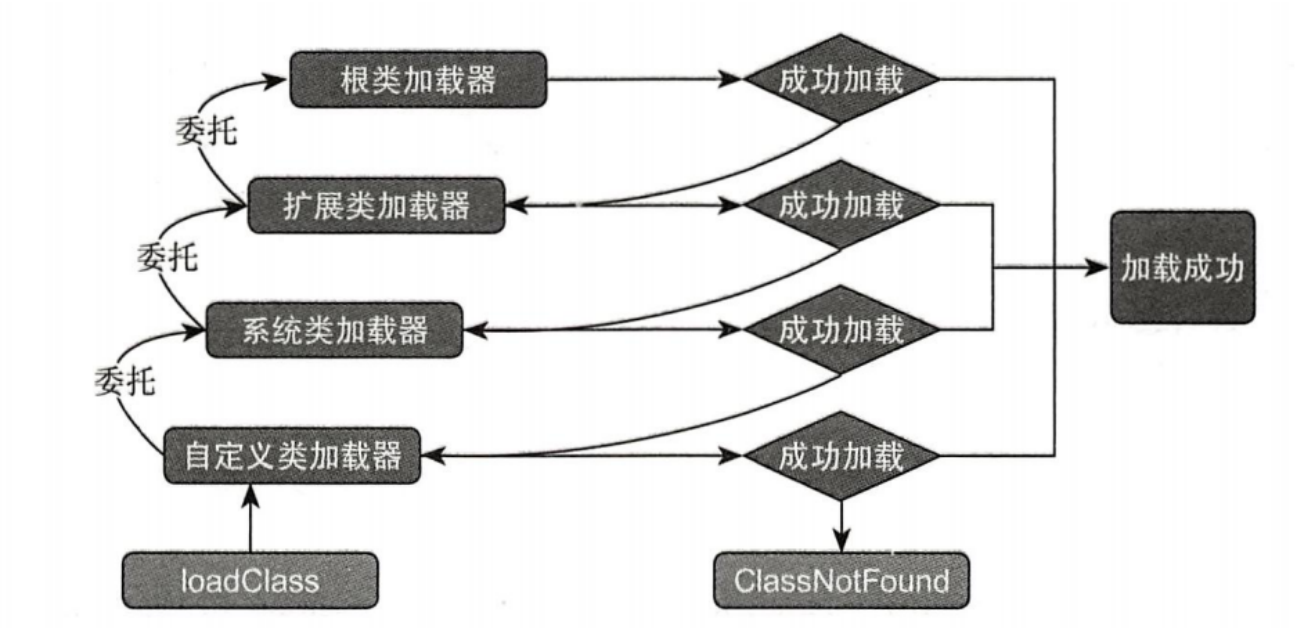
四、类加载器

1.JVM内置的三大类加载器



2.双亲委派机制

当一个类加载器调用loadClass后，它不会直接将其加载，而是交给当前类的父加载器尝试加载直到最顶层的父加载器，然后再依次向下加载。



2.1 如何在自定义的加载器绕过系统类加载器

- 方法一：直接将扩展Ext ClassLoader作为MyClassLoader的父加载器

- 方法二：指定MyClassLoader的父加载器为null

```
// loadClass有个判断
if (parent != null){
  clazz = parent.loadClass(name,...)     // 有父加载器让父类加载
}else{
  clazz = findBootstrapClassOrNull(name) // 没有父类加载器，直接让根加载器来加载
}
```

2.2 如何破坏双亲委派机制

```
JVM提供的双亲委派机制不是强制性的模型。
在自定义的类加载器的loadClass方法，修改调用的类加载器顺序。
        比如先使用自定义类加载器，如果加载失败再调用父类加载器
```

```
public class BrokerDelegateClassLoader extends ClassLoader {
...省略代码
@Override
protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
    //①
    synchronized (getClassLoadingLock(name)){
        //②
        Class<?> klass = findLoadedClass(name);
        //③
        if (klass == null)
        {
            //④
            if (name.startsWith("java.") || name.startsWith("javax"))
            {
                try
                {
                    klass = getSystemClassLoader().loadClass(name);
                } catch (Exception e)
                {
                    //ignore
                }
            } else
                {
                   //⑤
                    try
                    {
                        klass = this.findClass(name);
                    } catch (ClassNotFoundException e)
                    {
                        //ignore
                    }
                    //⑥
                    if (klass == null)
                    {
                        if (getParent() != null)
                        {
                            klass = getParent().loadClass(name);
                        } else
                        {
                            klass = getSystemClassLoader().loadClass(name);
                        }
                    }
                }
            }
            //⑦
            if (null == klass) {
                throw new ClassNotFoundException("The class " + name + " not  found.");
            }
            if (resolve){
                resolveClass(klass);
            }
            return klass;
        }
    }
    ...
    }
```

在上述代码中：

①根据类的全路径名称进行加锁，确保每一个类在多线程的情况下只被加载一次。

②到已加载类的缓存中查看该类是否已经被加载，如果已加载则直接返回。

③④若缓存中没有被加载的类，则需要对其进行首次加载，如果类的全路径以 java 和 javax 开头，则直接委托给系统类加载器对其进行加载。

⑤如果类不是以 java 和 javax 开头，则尝试用我们自定义的类加载进行加载。

⑥若自定义类加载仍旧没有完成对类的加载，则委托给其父类加载器进行加载或者系统类加载器进行加载。

⑦经过若干次的尝试之后，如果还是无法对类进行加载，则抛出无法找到类的异常。

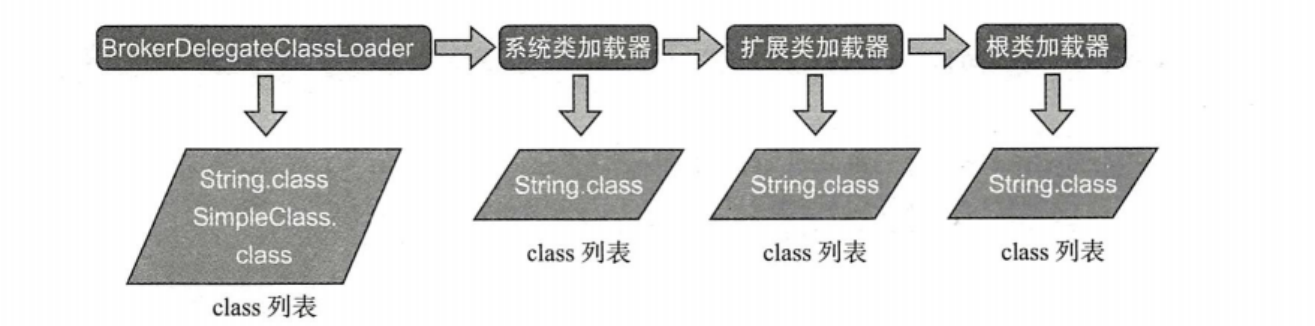
3.初始类加载器

同一个class实例在不同的类加载器命名空间下是不唯一的

运行时包：`类加载器的namespace + 类的全限定名称（路径）`

JVM规定：不同的运行时包下的类，是不能互相访问的。

> 问题：为什么com.wm包下，可以new String()。String是java.lang.String包下
>
> 答复：在类加载的过程中，所有参与的类加载器（即便没有亲自加载该类），也都会被标记为该类的初始类加载器。
> 比如String.class先后经过了：自定义加载器 -> 系统类加载器 -> 扩展类加载器 -> 根加载器，
> 所以每个加载器的class列表中都有String.class



4.类的卸载

类的实例如果没有其他地方引用，就会被GC回收。

那么该对象`在堆内存中class对象`以及`class在方法区的数据结构`何时回收

JVM规定：Class满足下面三个条件才被GC

- 该类的所有实例都已经被GC
- 加载该类的classLoader实例已经被回收
- 该类的class实例没有在其他地方被使用

5.双亲委派模型的缺陷

JDK核心类库提供了很多SPI，比如jdbc，具体的实现由第三方厂商来完成。

但是问题在于：java.lang.sql由JDK提供的，由根加载器加载（直接在根加载器的class列表中）。而第三方实现的驱动都是由系统类加载器进行加载（存放在系统类加载器的class列表中）。

由于双亲委派模型的限制，根加载器不可能加载得到第三方的具体实现。

> jdk只好提供一种不太好的设计：线程上下文加载器，根加载器需要委托子加载器去加载厂商提供的SPI具体实现。
> 这种行为打破了双亲委派机制模型的层次关系来逆向使用类加载器，实际上违背了双亲委派机制的一般性原则。

五、CPU cache模型和java内存模型

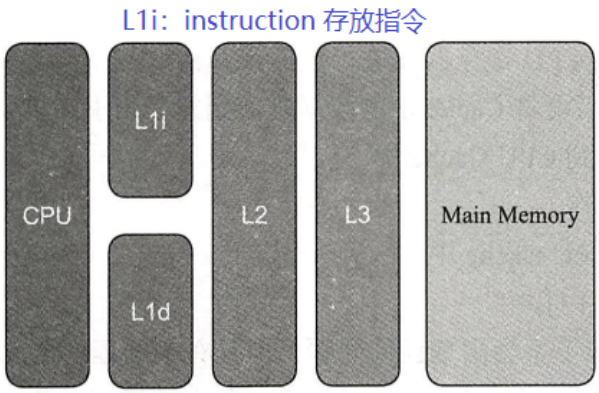
1.cpu cache模型

所有的运算操作都是由CPU寄存器(register)完成。而CPU的处理速度和内存的访问速度差距越拉越大。

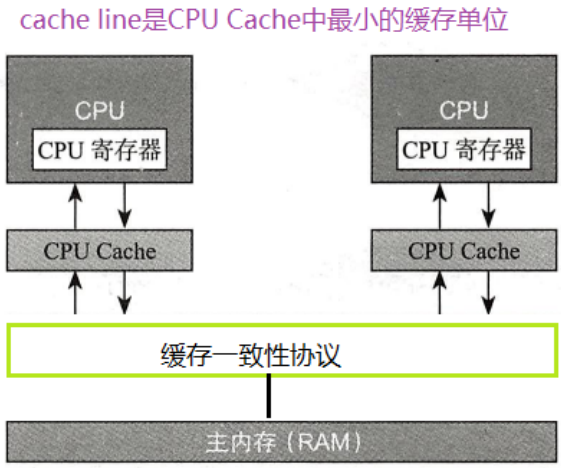
因此在CPU和主内存之间，就增加了缓存（L1，L2，L3）

而程序指令(instruction)和程序数据(data)的行为和热点分布差异很大，将L1又进行了划分：L1i和L1d

cache line是CPU Cache中最小的缓存单位。

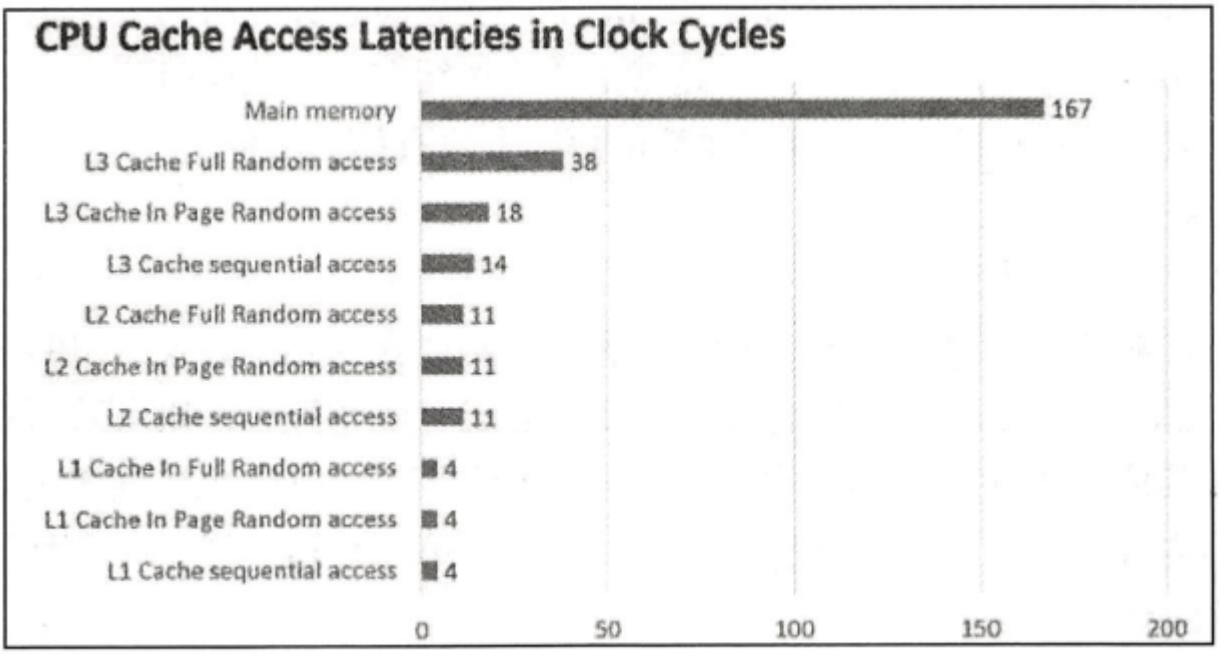


CPU Cache 模型



CPU 通过 Cache 与主内存进行交互

可以看到主内存和缓存的读写速度相差几倍甚至几十倍，更别说与CPU对比了。



Cache 与主内存访问速度对比图

程序运行过程，将数据从main memory中复制一份到CPU cache，CPU的寄存器计算时直接从缓存中读取和写入。
结束后再刷新到main memory中。

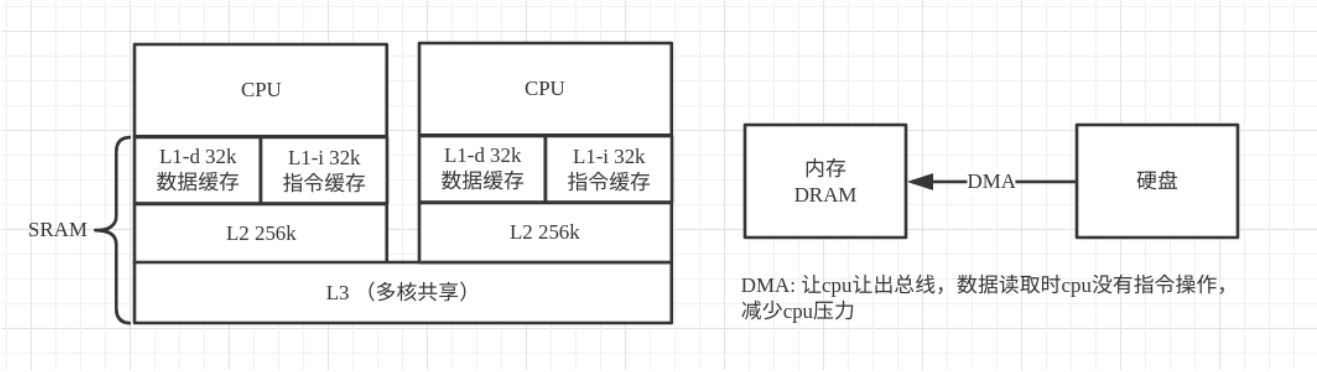
1.1 cpu(缓存)、内存、硬盘

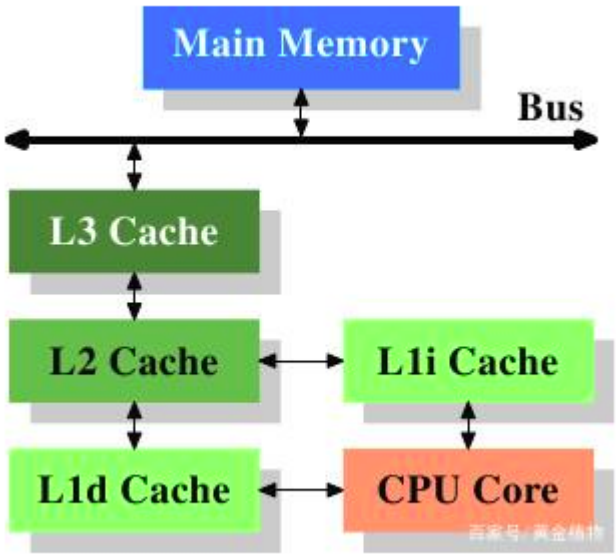
https://baijiahao.baidu.com/s?id=1598811284058671259&wfr=spider&for=pc

CPU：
　　CPU是中央处理器的简称，它可以从内存和缓存中读取指令，放入指令寄存器，并能够发出控制指令来完成一条指令的执行。但是CPU并不能直接从硬盘中读取程序或数据。

内存：
　　内存作为与CPU直接进行沟通的部件，所有的程序都是在内存中运行的。其作用是暂时存放CPU的运算数据，以及与硬盘交换的数据。也是相当于CPU与硬盘沟通的桥梁。只要计算机在运行，CPU就会把需要运算的数据调到内存中进行运算，运算完成后CPU再将结果传出来。

缓存：
　　缓存是CPU的一部分，存在于CPU里。由于CPU的存取速度很快，而内存的速度很慢，为了不让CPU每次都在运行相对缓慢的内存中操作，缓存就作为一个中间者出现了。有些常用的数据或是地址，就直接存在缓存中，这样，下一次调用的时候就不需要再去内存中去找了。因此，CPU每次回先到自己的缓存中寻找想要的东西（一般80%的东西都可以找到），找不到的时候再去内存中获取。
　　有了L3缓存后，基本只有5%的数据需要去内存读取了。





1.1.为什么cpu缓存比内存快？

cpu缓存：(SRAM : static random-access memory),电路结构复杂，占据面积大，由晶体管存储数据，只要通电数据就能保持，不需要刷新。因此速度很快。

内存：(DRAM : dynamic random-access memory),电路简单，占据面积小，由电容器存储数据，因此需要周期性的充电放电，否则数据可能丢失。充电放电的时间差导致其速度相对SRAM要慢。

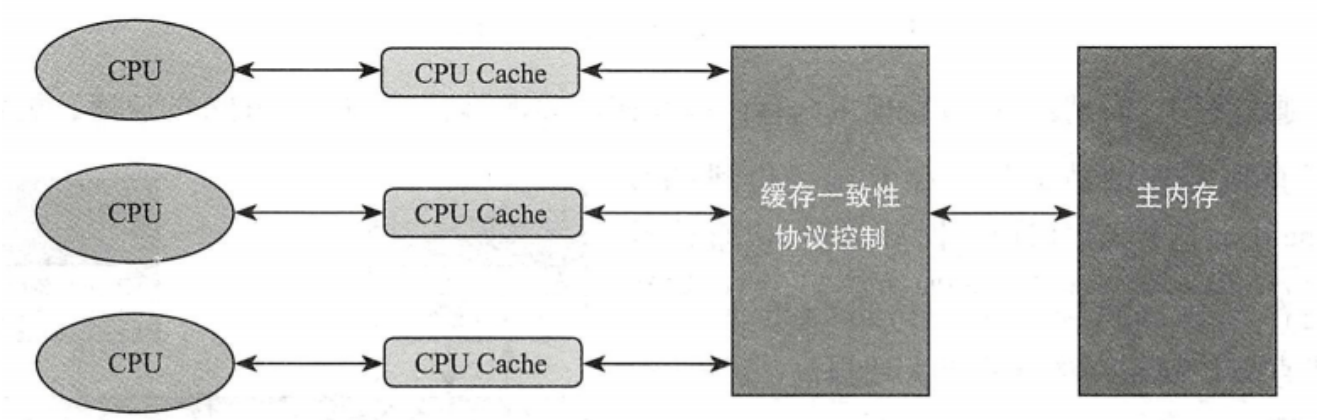
2.cpu cache的一致性问题

比如多个线程进行i++时，每个线程都有自己的工作内存（本地内存，对应CPU中cache）。计算时，都从main memory读取i到cpu cache，计算后再写入main memory　=> 导致不一致

```
解决方案：通过缓存一致性协议。最著名的是Intel的MESI协议
```

MESI协议实现思想：
当CPU操作cache中的数据时，如果发现是个共享变量（其他cpu cache中也有存在一个副本），那么：

- 读取操作：不做任何处理，只是将cpu cache中数据拷贝到寄存器中
- 写入操作：发出信号通知其他CPU将该变量的`cache line`置为无效状态。这样其他CPU在对该变量读取的时候，只能到main memory中重新读取。解决了一致性问题。



缓存一致性协议解决数据不一致问题

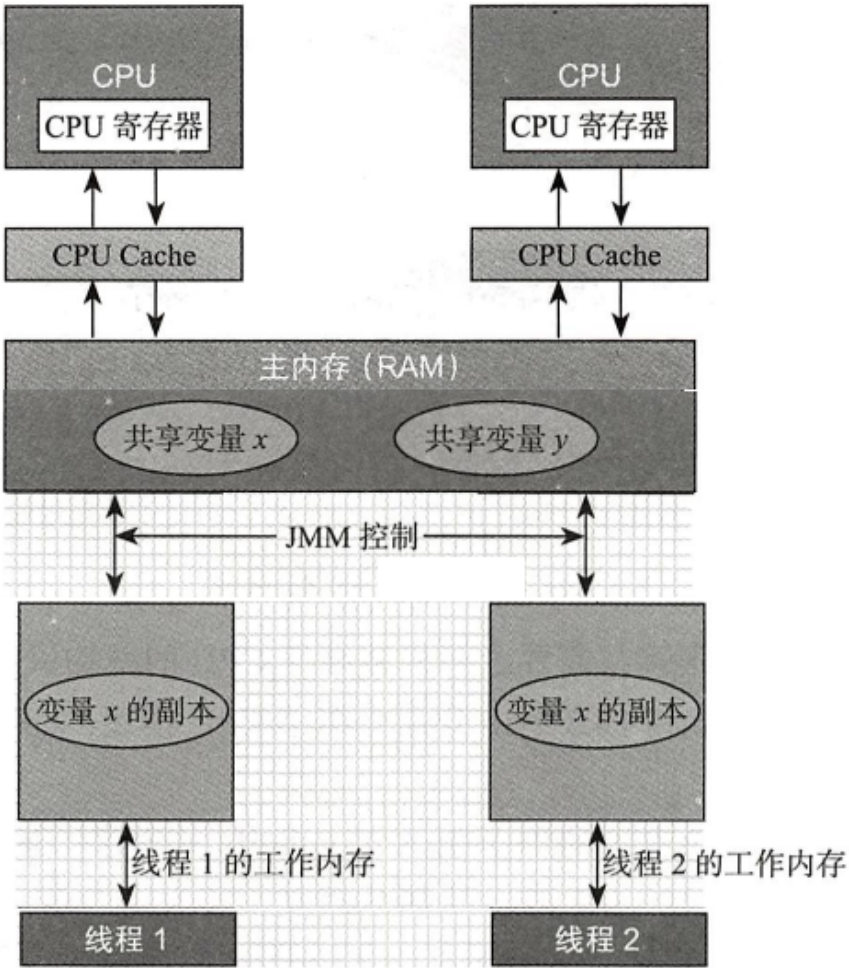
3.java内存模型（JMM）

```
面试连环炮：
JMM
--> 三大特性
--> volatile + 可见性(lock前缀指令+缓存一致性协议) + 有序性(内存屏障)
--> happens-before原则
--> volatile底层原理
```

java内存模型（Java Memory Mode）指定了JVM和主内存之间是如何工作的。

JMM定义了线程与main memory之间的关系：

- 共享变量存储在主内存中，每个线程都可以访问，每个线程都都自己的工作内存（本地内存）
- 工作内存只存储该线程对共享变量的副本
- 线程不能直接操作主内存，只有先操作工作内存才能写入主内存。

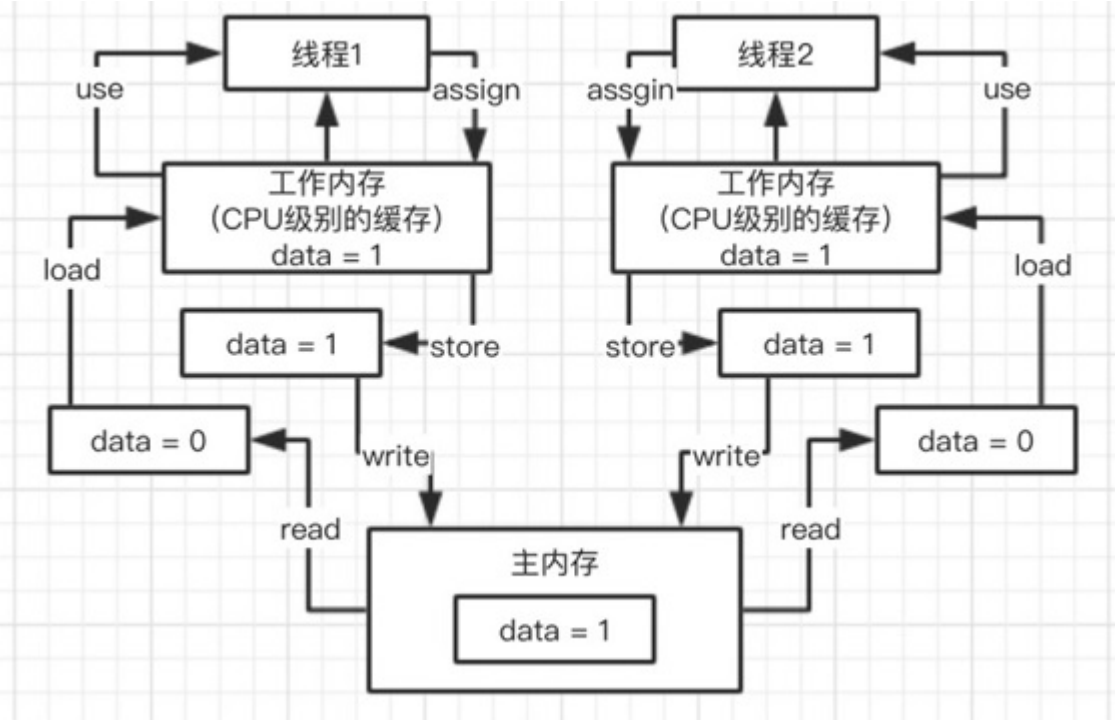


Java 内存模型（JMM）

工作内存和JMM一样，都是抽象的概念。

它涵盖了缓存、寄存器、编译器优化以及硬件等

java内存模型对应的六个指令：read、load、use、assign、store、write



3.1、并发编程的三大特性

- 原子性：一次或者多次操作中，所有操作要么都执行，要么都不执行。

  JMM只保证对基本数据和引用数据类型的读取和赋值都是原子性的。
  如果想使得代码片段具备原子性，需要使用关键字synchronized或者java.util.concurrent并发包中的lock。
  volatile不保证原子性，synchronized保证原子性。
  AtomicInt等原子类型的变量保证原子性
  两个原子性的操作结合在一起不一定时原子性的。比如i++

- 有序性

  JVM会对代码进行指令重排。但是在并发情况下，有些重排可能会导致错误

  ```
  if (!flag) {
    context = new Context(); // ①
    flag = true;     // ②
  }
  // 如果① ② 重排，另外一个线程可能就无法new了
  ```

- 可见性

  当一个线程对共享变量进行了修改，其他线程能立刻看到修改后的最新值。

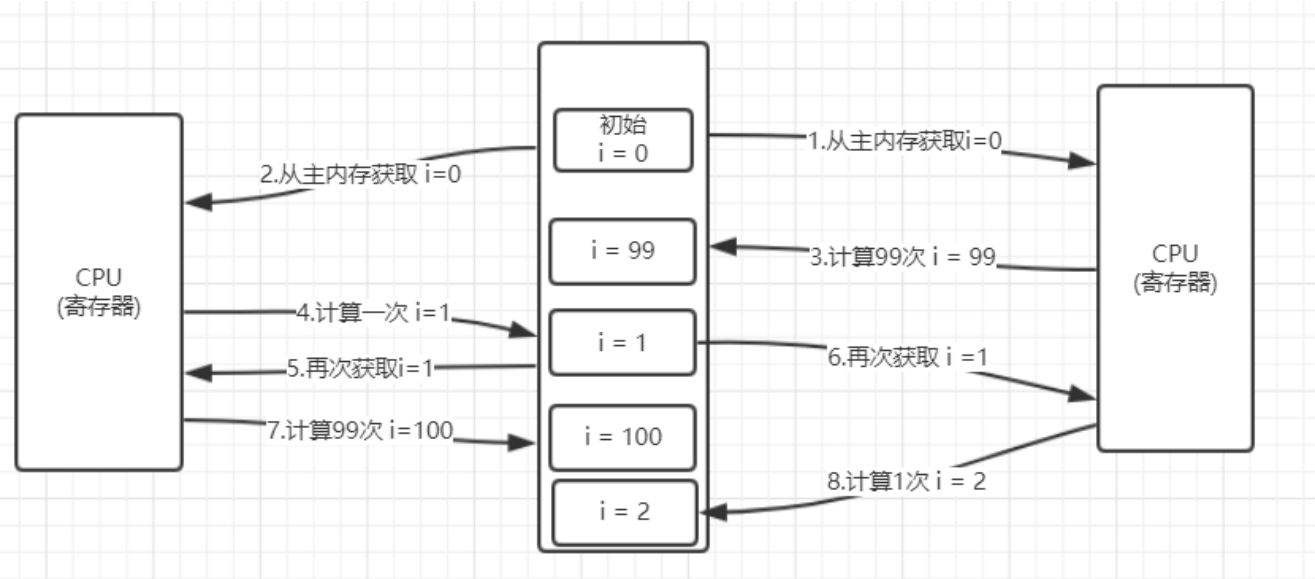
	原子性	有序性	可见性
synchronized	√	√	√
并发包的Lock（AQS）	√	√	√
volatile	×	√	√

面试题

i++为何不是原子性操作

在出来i++时，有三个步骤：从主内存获取i到工作内存 --> i加1 --> 刷新到主内存。
比如当i =100时，两个线程都进入。此时主内存i = 100，线程A，B的工作内存都是101，刷新到主内存也都是101.但是进行了两次i++。

i++在两个线程分别执行100次，最大值和最小值分别多少？（2~200）



3.2、JMM如何保证原子性、可见性、有序性

1.JMM与原子性

synchronized或者java.util.concurrent并发包中的Lock

JMM只保证对基本数据和引用数据类型的读取和赋值都是原子性的。
如果想使得代码片段具备原子性，需要使用关键字synchronized或者java.util.concurrent并发包中的lock。
volatile不保证原子性，synchronized保证原子性。
AtomicInt等原子类型的变量保证原子性
两个原子性的操作结合在一起不一定时原子性的。比如i++

2.JMM与可见性、有序性

JMM有三种方式保证可见性和有序性：

1. volatile关键字修饰变量保证可见性和有序性

   对volatile修改的变量进行修改时，会导致其他工作内存中共享变量失效，其他线程获取的时候，必须从主内存获取。

   volatile会禁止JVM和处理器对其修饰的字符进行指令重排操作。

2. synchronized关键字保证可见性和有序性 （**主要是通过串行化执行而实现的**）

   它保证同一时间只有一个线程获得锁。然后执行同步方法，并`保证在锁释放前，将变量的修改刷新到主内存中`。

   synchronized采用了同步的机制，执行和单线程一样，保证最终结果的顺序性。

3. JUC下的Lock.lock( )能保证可见性和有序性

   实现原理和synchronized一样。只有一个线程获得锁，并且保证unlock时，修改刷新到主内存中去。

4.volatile

volatile不保证原子性，但是保证可见性和有序性（禁止指令重排）

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

- 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

  > 第一步：使用volatile关键字的话，当线程2对变量A进行修改时，会导致线程1的工作内存中缓存变量A的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行(Cache Line)无效）
  > 第二步：由于线程1的工作内存中缓存变量stop的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

- 禁止jvm和处理器对volatile修饰的变量进行指令重排序。

volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性

案例：

```
public class test implements Runnable {
    public volatile int value = 0;  // 方法一：public  AtomicInteger value = new
AtomicInteger(0);

    @Override
    public void run() {
        increse();
    }

    public void increse() {         // 方法二：加上synchronized关键字或者锁
        for (int i = 0; i < 10000; i++) {
            try {
                value++;            // value.getAndIncrement();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName() + ": value = " + value);
    }

    public static void main(String[] args) throws Exception {
       test test1 = new test();
       Thread t1 =  new Thread(test1, "thread-1");
       Thread t2 =  new Thread(test1, "thread-2");
       t1.start();
       t2.start();
       t1.join();   // 保证线程t1和t2执行结束后，才执行下面的语句
       t2.join();
       System.out.println(Thread.currentThread().getName() + ": value = " + test1.value);
    }
}
```

上面这段程序执行时，两个线程计算，最终的结果很少是20000.因为value虽然是用volatile修饰的，
但是它并不能保证原子性。想要实现原子性操作。采用AtomicInteger或者采用synchronized关键字。

4.2 volatile实现原理

```
lock指令+缓存一致性协议     => volatile的可见性
内存屏障                   => volatile的有序性
```

被volatile修饰的变量会有一个`lock;`的前缀，相当于一个`内存屏障`。lock；的作用为：

- 指令重排时，禁止前面的代码重排到后面，禁止后面的代码重排到前面 （有序性）
- 强制工作内存中变量的修改刷新到main memory中。并使得其他线程的工作内存（CPU cache）中缓存数据失效 （可见性）

```
volatile修饰的变量执行写操作时，JVM会发送一条"lock前缀"的指令给CPU，CPU在计算完之后会立即将这个值写回主内存，同时因为MESI缓存一致性协议，所以各个CPU都对总线BUS进行嗅探：自己本地缓存中的数据是否被别的线程修改了？

如果被修改了，CPU会将自己本地缓存的数据(cache line)过期掉，然后该CPU执行读取操作时，从主存重新读取。
```

java编译器、指令器对代码重排时，要遵循happens-before原则，

其中有一条就是与volatile相关：

```
volatile变量规则：对一个volatile变量的写操作先行发生于后面的读操作。必须保证先写再读。
所以编译器不会对其指令重排的！
```

4.3 volatile使用场景

- 维护中间状态，一旦修改别的线程要立马感知到。
- CopyOnWrite思想。数据有的线程读、有的线程写，避免加读写锁，采用该方法。

比如kafka写数据的时候，都是写入一个副本的map，然后赋值给map。这样读线程能立刻感知到。

```
// 这个map是核心的，因为用volatile修饰了。
// 只要把最新的数组对他赋值，其他线程立马可以看到最新的数组
private volatile Map<K, V> map;

@Override
public synchronized V put(K k, V v) {
    Map<K, V> copy = new HashMap<K, V>(this.map);
    V prev = copy.put(k, v);
    this.map = Collections.unmodifiableMap(copy);
    return prev;
}

@Override
public synchronized void putAll(Map<? extends K, ? extends V> entries) {
    Map<K, V> copy = new HashMap<K, V>(this.map);
    copy.putAll(entries);
    this.map = Collections.unmodifiableMap(copy);
}
```

5 Synchronized vs volatile

Synchronized和ReenTrantLock的对比

1. 可重入性。都是可重入锁，在锁的计数器上+1，释放锁的时候-1。
   实现方式是不一样的。
   Synchronized是基于锁中的monitor对象，里面也有一个count计数，重入的时候count++。
   ReenTrantLock是基于AQS实现的，在AQS内部有个state记录线程持有锁的计数，thread记录当前持有锁的线程。
2. 公平性。
   Synchronized是非公平锁，通过自旋的方式获得锁。（如何实现的？？？）
   ReenTrantLock( )非公平锁， ReenTrantLock(True)是公平锁，
3. 实现方式。Synchronized是基于JVM实现的，ReenTrantLock是基于JDK实现的
4. 性能。在Synchronized优化前，性能较差。优化后（轻量级锁（自旋锁）、偏向锁）性能差不多。相同环境下，官方更建议使用Synchronized。
5. volatile作用于变量，synchronized作用于方法和代码块
6. volatile不会使得线程阻塞，而synchronized会导致线程进入阻塞状态。
7. 原子性。volatile不能满足原子性

ReenTrantLock独有的能力:

1. ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
2. ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要指定的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。
3. ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。这弥补了Synchronized一个其中缺陷。

六、单例设计模式

1.懒汉式

问题：多线程下可能创建多个instance

```
public final class SingletonDemo {  // 用final修饰该类，不能被继承

    private SingletonDemo() { }

    private static SingletonDemo instance = null;

    // 懒汉式
    // 缺点：多线程下可能创建多个instance
    public static SingletonDemo getInstance() {
        if (instance == null)  // 多个线程进入判断
            instance = new SingletonDemo();
        return instance;
    }
    // 懒汉式 + 同步方法Synchronized
    // 缺点：它是串行化执行，如果有 10000次调用，很慢
    public static synchronized SingletonDemo getInstance() {
        if (instance == null)
            instance = new SingletonDemo();
        return instance;
    }
}
```

2.double-check + volatile

只在第一次进行实例化的时候synchronized修饰，进行同步。性能提高很多

```
public final class SingletonDemo {
    Socket socket;
    Connection conn;

    private SingletonDemo() {  // SingletonDemo中需要实例化socket和conn这两个资源
        this.socket;
        this.conn;
    }

    private static SingletonDemo instance = null;   // volatile修饰，防止socket、conn空指针异常

    public static SingletonDemo getInstance() {
        if (instance == null) {
            synchronized (SingletonDemo.class) {
                if (instance == null) {
                    instance = new SingletonDemo();
                }
            }
        }
        return instance;
    }
}
```

缺点：除了要实例化instance，还需要实例化socket和conn这两个资源。由于指令重排的原因，他们并没有先后关系。

线程A实例化instance后，socket或conn还没有实例化完成。线程B判断instance != null，运行时socket或conn由于没有实例化完成，会抛出空指针异常。

方法：volatile修饰instance，防止指令重排。

3.枚举方式（很高效）

枚举不允许被继承（不用final修饰），并且是线程安全的，只会被实例化一次。

但是枚举本身不能实现懒加载，需要通过内部枚举的形式实现懒加载。

```
public class SingletonDemo {

    private SingletonDemo() { }

    private enum EnumHolder {
        INSTANCE;
        private SingletonDemo instance;
        // 枚举的构造方法
        EnumHolder() {
            instance = new SingletonDemo();
        }
        private SingletonDemo getInstance() {
            return instance;
        }
    }

    public static SingletonDemo getInstance() {
        return EnumHolder.INSTANCE.getInstance();
    }
}
```

七、多线程设计架构模式

1.single thread execution模式

类似过安检的案例。采用synchronized关键字，保证串行化执行，同一时间只有一个线程在操作共享变量。

2.不可变对象设计模式

类似string，每次修改都是返回新的对象，线程没有机会修改原来的对象，进而达到free lock的状态。

比如ArrayList的stream在多线程情况下也是线程安全的，因为每次都是生成新的list

3.Future设计模式

其实就是定义一个线程池，执行任务。并将计算结果写入Future对象中，调用future.get()会阻塞获得结果，future。idDone()可以计算是否完成。jdk1.8中还有isCancelled，get(timeout)等方法。

4.Guarded suspendion设计模式

Balking：当到达临界值就放弃。

Guarded suspendion是`确保挂起`的意思，当到达临界值就暂时挂起。

比如BlockingQueue的take和put操作。当队列为空，take

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)              // 条件满足就挂起
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)   // 条件满足就挂起
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}
```

5.线程上下文设计模式

其实就是定义一个Map，key值为当前线程，value为对应的上下文ActionContext。这样就保证了线程上下文之间的独立性，而且不用考虑ActionContext的线程安全性，因为只有一个线程访问。

但是这种方式会导致内存泄漏：

Map是当前线程作为key，线程结束后，Map中的Thread实例不会得到释放，对应的value也不会释放。时间长了就会导致内存泄漏（memory leak）。可以通过soft reference或者weak reference等引用类型

实际开发中，其实是利用ThreadLocal实现线程上下文的设计

5.1、ThreadLocal的原理

ThreadLocal为了是变量成为每个线程独有，不再是共享变量。对变量的修改不会被其他线程获得。

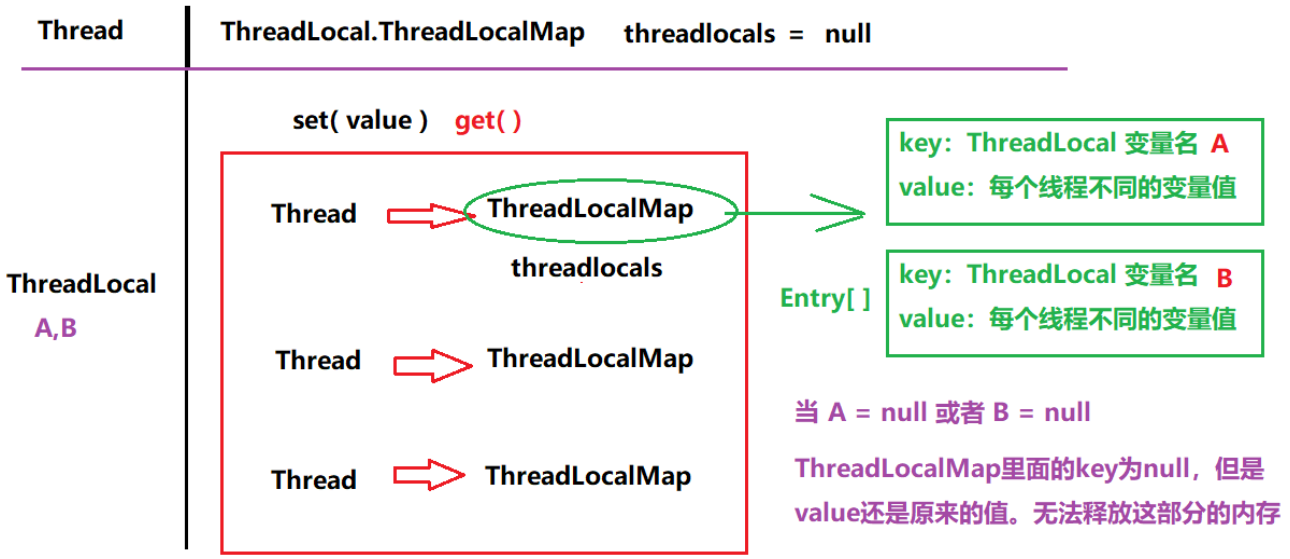
```
// ThreadLocal源码set方法
public void set(T value) {
    Thread t = Thread.currentThread();  // 获取当前线程
    ThreadLocalMap map = getMap(t);     // 获取当前线程的ThreadLocalMap，默认值为null
    if (map != null)             // 如果ThreadLocalMap为null，则创建一个。因此它是lazy加载，默认长度为16
        map.set(this, value);
    else
        createMap(t, value);
}

-------------- ThreadLocalMap的定义 --------------
    static class ThreadLocalMap {
        /**
         * The entries in this hash map extend WeakReference, using
         * its main ref field as the key (which is always a
         * ThreadLocal object).  Note that null keys (i.e. entry.get()
         * == null) mean that the key is no longer referenced, so the
         * entry can be expunged from table.  Such entries are referred to
         * as "stale entries" in the code that follows.
         */
        static class Entry extends WeakReference<ThreadLocal<?>> {
            Object value;
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
        // 初始容量为16
        private static final int INITIAL_CAPACITY = 16;
        private Entry[] table;
    }
```

ThreadLocalMap存储的是Entry数组，每个Entry对象都是以ThreadLocal对象作为key值的。
Entry是WeakReference的子类，为了能够在JVM发生垃圾回收事件时，能自动回收防止内存溢出问题的出现。

5.2、ThreadLocal的内存泄漏问题

ThreadLocal在ThreadLocalMap中，被Entry中的Key弱引用的`WeakReference`，因此如果ThreadLocal没有外部强引用来引用它。ThreadLocal会在下次JVM垃圾收集时被回收。这个时候就会出现Entry中Key已经被回收，出现一个null Key的情况，外部读取ThreadLocalMap中的元素是无法通过null Key来找到Value的。

因此如果当前线程的生命周期很长，一直存在，那么其内部的ThreadLocalMap对象也一直生存下来，这些null key就存在一条强引用链的关系一直存在：Thread --> ThreadLocalMap-->Entry-->Value，这条强引用链会导致Entry不会回收，Value也不会回收，但Entry中的Key却已经被回收的情况，造成内存泄漏。`比如在for循环中大量创建threadlocal变量`



JVM团队做了一些措施来保证ThreadLocal尽量不会内存泄漏：
在ThreadLocal的get()、set()、remove()方法调用的时候会清除掉线程ThreadLocalMap中所有Entry中Key为null的Value，并将整个Entry设置为null，利于下次内存回收。

```
private Entry getEntry(ThreadLocal key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);    // 删除key = null所对应的值
}
```

但是即使这样还是有可能内存泄漏

1.使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致的内存泄漏。
2.分配使用了ThreadLocal，但是又不再调用get()、set()、remove()方法。
即使key为null了，无法删除key为null所对应的value。那么就会导致内存泄漏。

5.3、什么是WeakReference、SoftReference

1. 强引用(StrongReference)：

   强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它

2. 软引用(SoftReference)：

   如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

3. 弱引用(WeakReference)：

   弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

   `软引用的生命周期为：下一次GC之前`

当想引用一个对象，但是这个对象有自己的生命周期，又不想介入这个对象的生命周期，这时候就用`弱引用`。

弱引用相当于划分了特别的区域，某个对象只有弱引用的时候，会被回收。

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	在内存不足时	对象缓存	内存不足时终止
弱引用	在垃圾回收时	对象缓存	gc运行后终止

5.4、基于软引用 + 双向链表 + hash表 实现LRUCache

软引用：将存储的value，采用new SoftReference<>(value))形式，防止内存溢出

双向链表：存储key值，并修改顺序

hash表：存储key-value

```
public class Reference {
    private final byte[] data = new byte[2<<19];  // 大约1M
}

public interface CacheLoader<K, V> {
    V load(K k);
}
```

```java
/*
采用双向链表 + hash表来实现LRU cache
*/
public class LruCache<K, V> {
    // keyList 最近最少使用的key存在最前面
    private LinkedList<K> keyList = new LinkedList();

    // private Map<K, SoftReference<V>> cache = new HashMap<>();
    // 将所有存储对象都转成软引用，保证内存不足时进行GC回收。
    private Map<K, V> cache = new HashMap<>();

    private int capacity;

    private CacheLoader<K, V> cacheLoader;

    public LruCache(int capacity, CacheLoader<K, V> cacheLoader) {
        this.capacity = capacity;
        this.cacheLoader = cacheLoader;
    }

    public void put(K key, V value) {
        // 如何cache满了，删除最近最少使用的
        if (cache.size() >= capacity) {
            K k = keyList.removeFirst();
            cache.remove(k);
        }
        // 如何keyList有这个key，删除key，在尾部添加该key
        if (keyList.contains(key)) {
            keyList.remove(key);
        }
        keyList.addLast(key);

        // cache.put(key, new SoftReference<>(value));
        cache.put(key, value);
    }

    public V get(K key) {
        // 如何keyList有这个key，删除key，在尾部添加该key
        V value;
        if (keyList.contains(key)) {
            keyList.remove(key);
            keyList.addLast(key);
            //  value = cache.get(key).get(); 获取软引用的value
            value = cache.get(key);
        } else {
            value = cacheLoader.load(key);
            this.put(key, value);
        }
        return value;
    }

    @Override
    public String toString() {
```

```
        return keyList.toString();
    }
}
```

```
public static void main(String[] args) {
    // 这是接口的匿名实现，与lambda实现效果是一样的。
    // 因为CacheLoader就一个方法，lambda表达式的实现表示load方法的返回结果是：new Reference()
    LruCache<String, Reference> test = new LruCache<>(5, new CacheLoader<String, Reference>() {
        @Override
        public Reference load(String s) {
            return new Reference();
        }
    });
    LruCache<Integer, Reference> lruCache = new LruCache<>(200, key -> new Reference());
    for (int i = 0; i < Integer.MAX_VALUE; i++) {
        try {
            lruCache.get(i);  // 会一直往lruCache中添加key=i，value = new Reference()的数据
            TimeUnit.MILLISECONDS.sleep(1000);
            System.out.println("The " + i + " Reference is  stored in cache.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

启动的配置参数为：-Xmx128M -Xms64M -XX:+PrintGCDetails（打印GC信息）
[GC (Allocation Failure) [PSYoungGen: 15875K->2200K(18944K)] 15875K->12448K(62976K), 0.0029167
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

6.Latch（门阀）设计模式

类似一个计数器的功能，当完成指定的任务个数，才继续执行

```java
public class LatchTest {
    public static void main(String[] args) throws InterruptedException {
        // 设置计数为3
        CountDownLatch latch = new CountDownLatch(3);
        IntStream.range(0, 3).forEach(i -> {
            new Thread(() -> {
                try {
                    System.out.println(Thread.currentThread().getName() + " is doing job");
                    TimeUnit.SECONDS.sleep(i);
                    latch.countDown();      // 每个线程完成后，执行countDown方法看，计数减1
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }, "Thread-" + i).start();
        });
        System.out.println("等待任务全部完成。。。");
        latch.await();                      // 等待latch的计数为0，所有任务都完成
        System.out.println("success");
    }
}
```