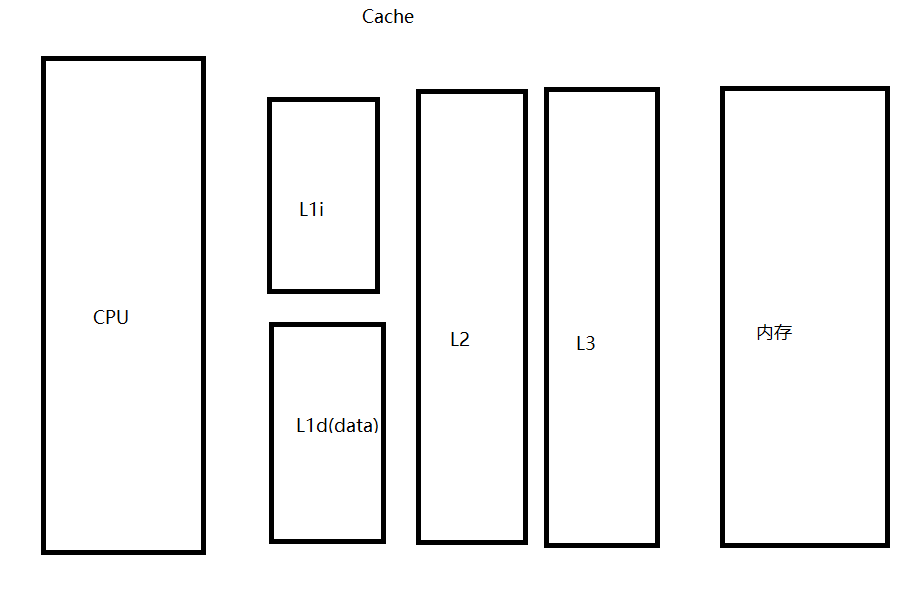
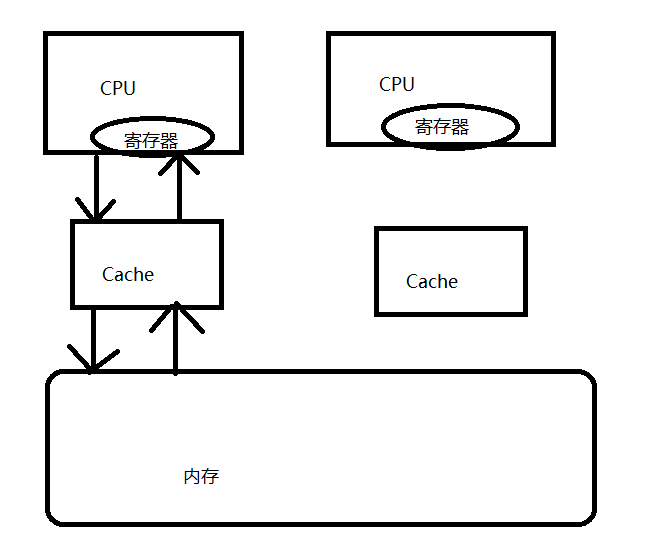
# Volatile关键字实现原理

## 机器硬件CPU与JMM

1. CPU Cache模型

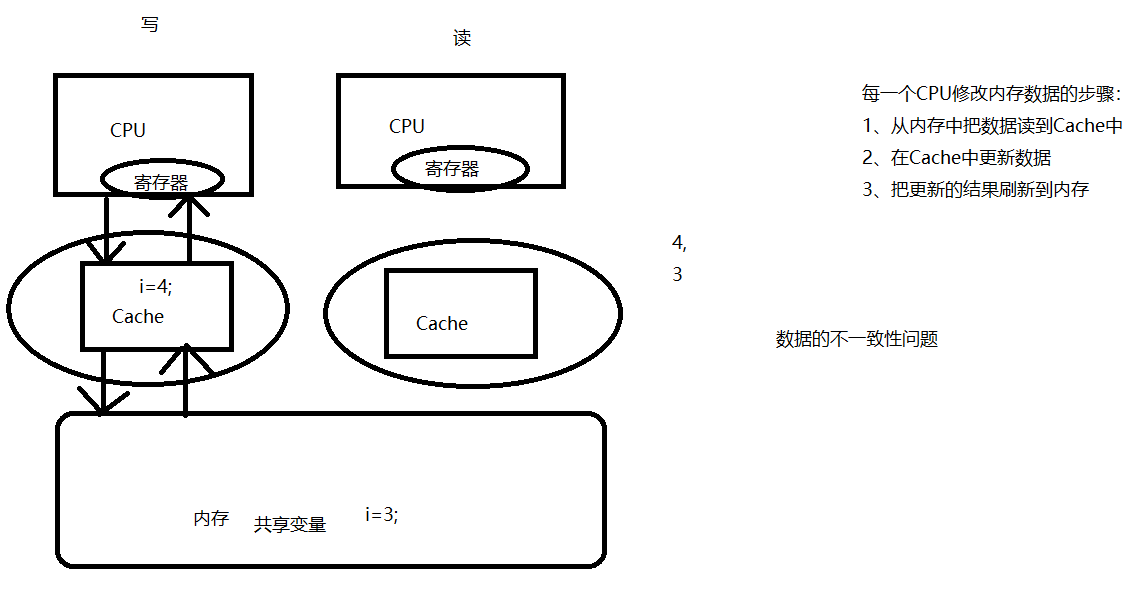
L1i检查数据，L1d数据存储





程序的局部执行原理

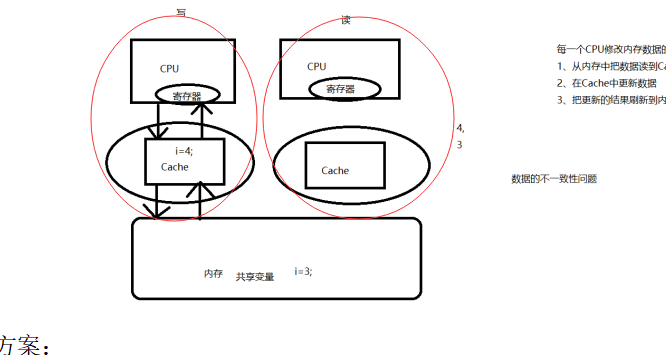
1. CPU缓存的一致性问题



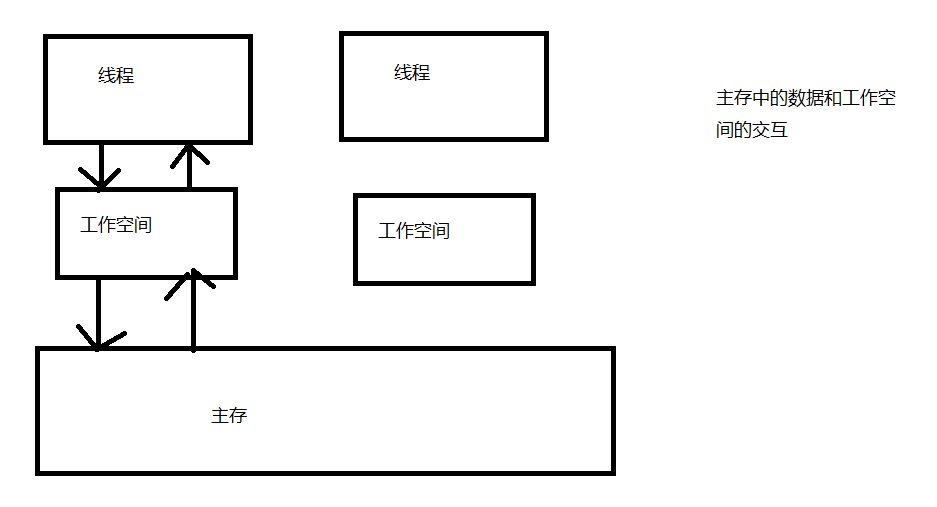
解决方案：

1. 总线加锁BUS（对单个CPU加锁，粒度太大）
2. MESI（缓存一致性协议）
   1. 读操作：不做任何事情，把Cache中的数据读到寄存器
   2. 写操作：发出信号通知其他的CPU讲该变量的Cache line置为无效，其他的CPU要访问这个变量的时候，只能从内存中获取。

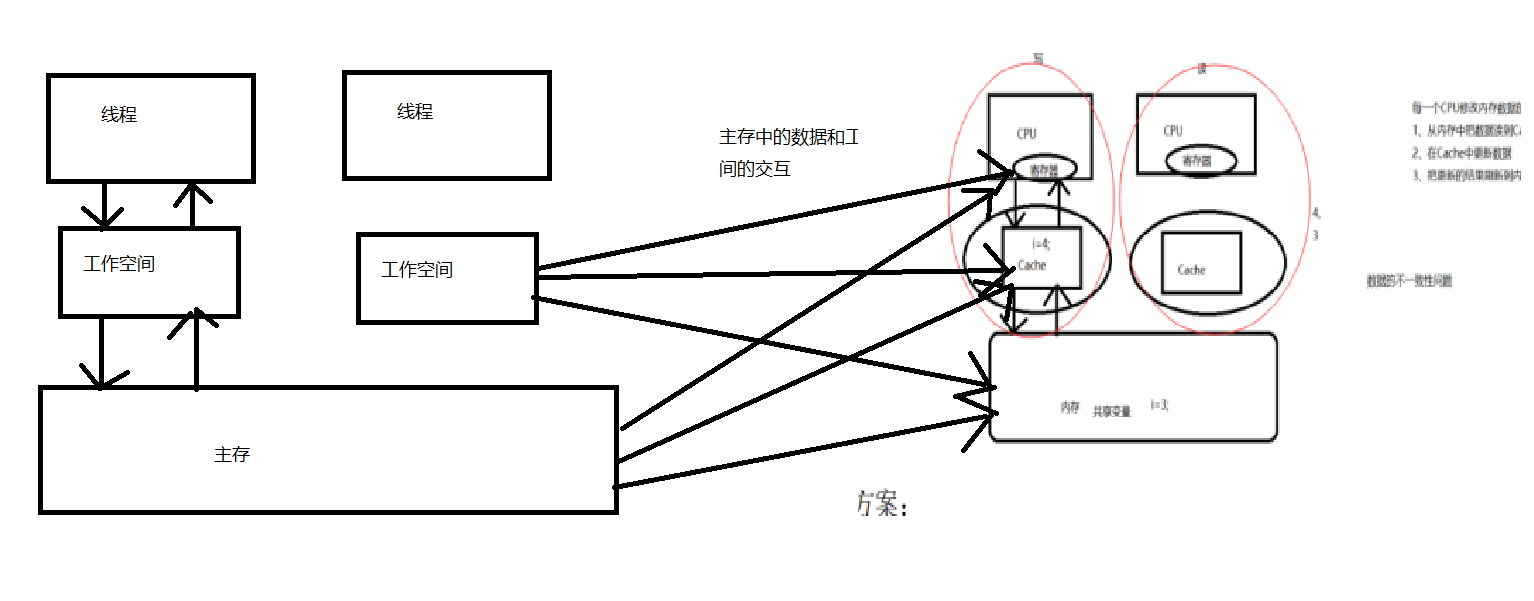
Cache line CPU的cache中会增加很多的Cache line。



1. Java内存模型



1. 主存中的数据所有线程都可以访问（共享数据）。
2. 每个线程都有自己的工作空间，（本地内存）（私有数据）。
3. 工作空间数据：局部变量、内存的副本。
4. 线程不能直接修改内存中的数据，只能读到工作空间来修改，修改完成后刷新到内存。



## Volatile关键字的语义分析

volatile作用：让其他线程能够马上感知到某一线程多某个变量的修改。

1. 保证可见性

对共享变量的修改，其他的线程马上能感知到，不能保证原子性 ， 读、写、（i++非原子性）

1. 保证有序性

重排序（编译阶段、指令优化阶段）

输入程序的代码顺序并不是实际执行的顺序

重排序后对单线程没有影响，对多线程有影响

Volatile

Happens-before原则

volatile规则：

对于volatile修饰的变量：

1. volatile之前的代码不能调整到他的后面
2. volatile之后的代码不能调整到他的前面（as if seria）
3. 霸道（位置不变化）

Int i=0;

Int a=3;

Int b=5;

Volatile Int j=3;

Int i=0;

Int a=3;

Int b=5;

Int m=i+j;

I++;

J++;

1. volatile的原理和实现机制(锁、轻量级)

HSDIS工具 --反编译---汇编

Java --class---JVM---》ASM文件

Volatile int a ; -->>Lock :a

## Volatile的使用场景

1. 状态标志（开关模式）

public class ShutDowsn extends Thread{

private volatile boolean started=false;

@Override

public void run() {

while(started){

//执行任务

dowork();

}

}

public void shutdown(){

started=false;

}

}

1. 双重检查锁定（double-checked-locking）

DCL（7种，单例模式）

public class Singleton {

private volatile static Singleton instance;

public static Singleton getInstance(){

if(instance==null){

synchronized (Singleton.class){

instance=new Singleton();

}

}

return instance;

}

}

1. 需要利用顺序性

## volatile与synchronized的区别

1. 使用上的区别

Volatile只能修饰变量，synchronized只能修饰方法和语句块。

1. 对原子性的保证

synchronized可以保证原子性，Volatile不能保证原子性。

1. 对可见性的保证

都可以保证可见性，但实现原理不同

Volatile对变量加了lock，synchronized使用monitorEnter和monitorexit jvm指令（代码块），monitor。

1. 对有序性的保证

Volatile能保证有序，synchronized可以保证有序性，但是代价（重量级）并发退化到串行。

1. 其他

synchronized引起阻塞，Volatile不会引起阻塞。