# Java并发编程与高并发解决方案

## 前期准备

### 1.1并发编程初体验

|  |
| --- |
| G:\concurrent\_programming>mvn -B archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DgroupId=com.byf -DartifactId=concurrent |

实现一个计数器

|  |
| --- |
| @Slf4j **public class** CountExample {  **private static int** *threadTotal* = 200;  **private static int** *clientTotal* = 5000;   **private static long** *count* = 0;   **public static void** main(String[] args) {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(*threadTotal*);  **for** (**int** index=0; index<*clientTotal*; index++){  exec.execute(() -> {  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (Exception e) {  ***log***.error(**"exception"**, e);  }  });  }  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  }   **private static void** add() {  *count*++;  } } |

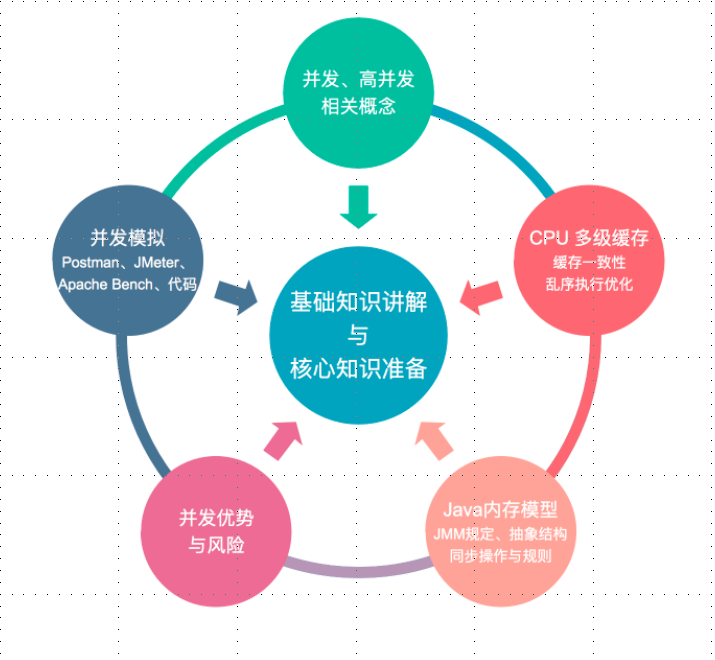
### 1.2并发与高并发基本概念

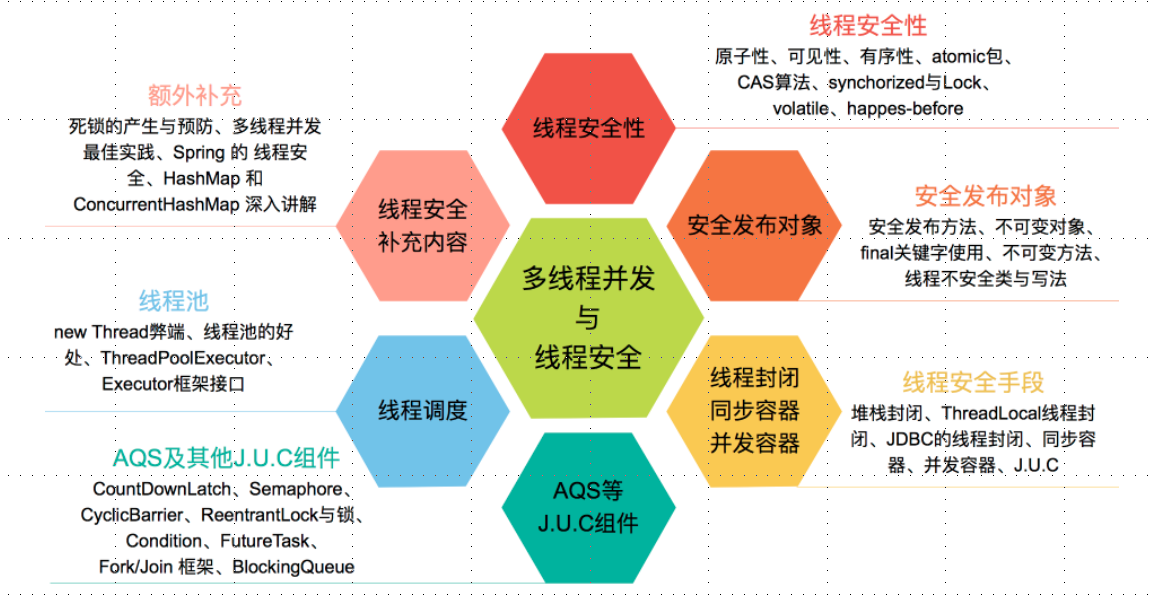
·并发：单核，多个线程交替执行；多核，每个线程分配到一个处理器。

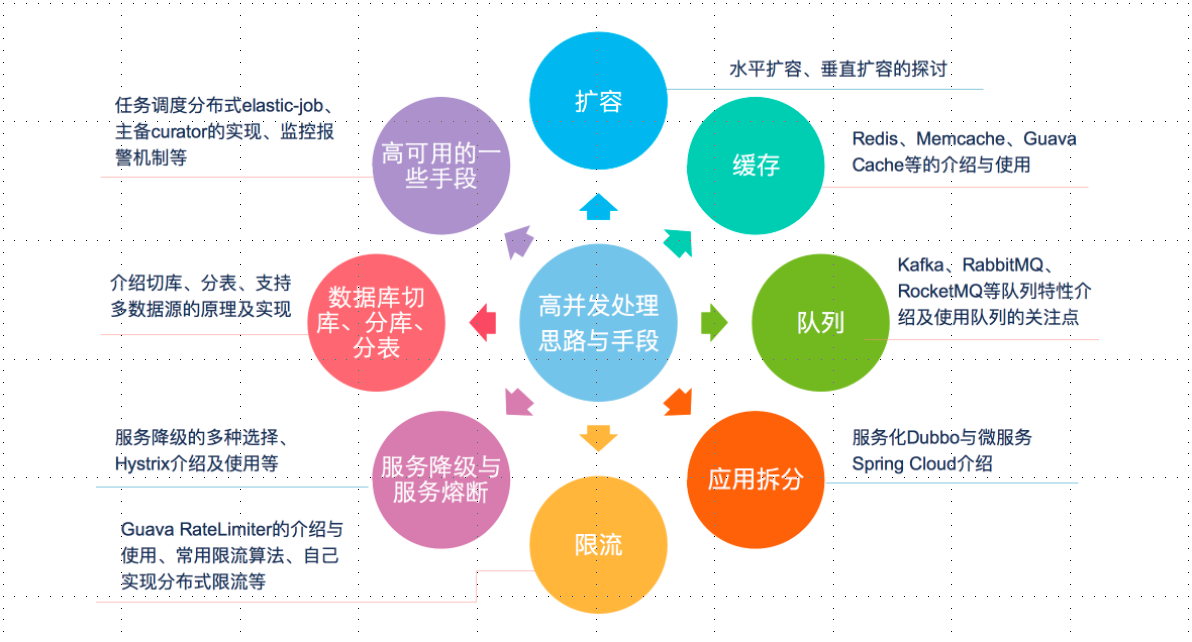
·高并发：互联网分布式架构设计必须考虑的因素，通常指，通过设计保证系统能够同时并行处理多个请求。

·并发：多个线程操作相同资源，保证线程安全，合理使用资源；

·高并发：服务能够同时处理很多请求，提高程序性能。



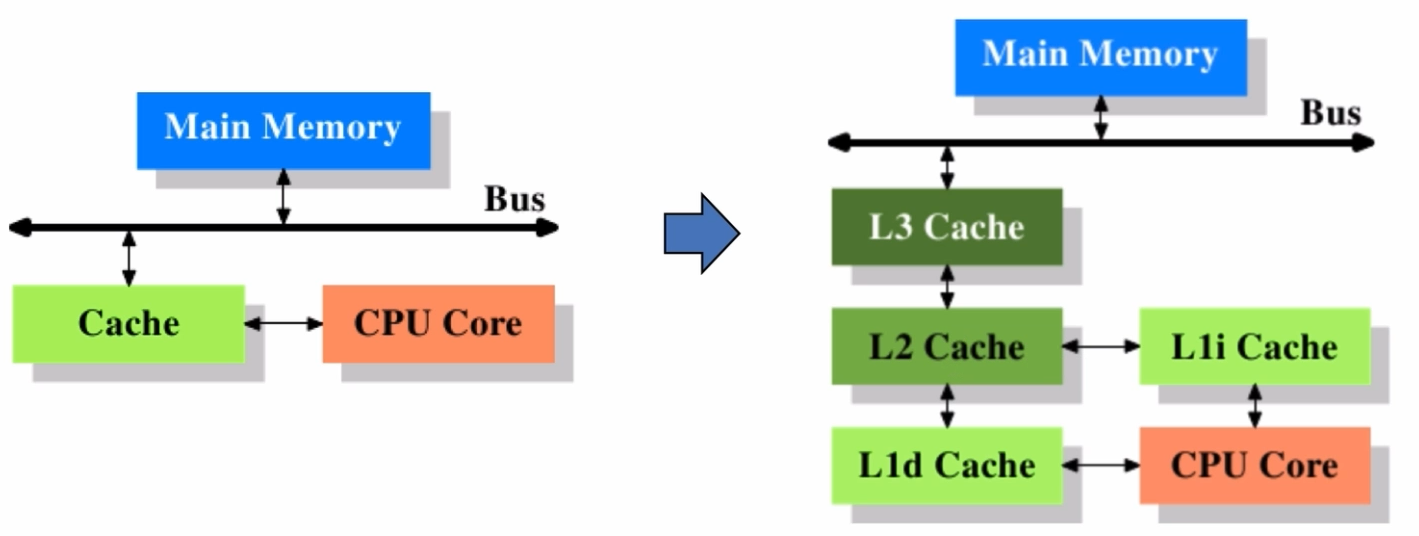


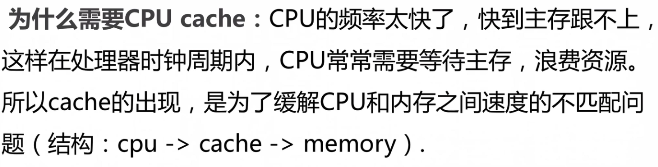


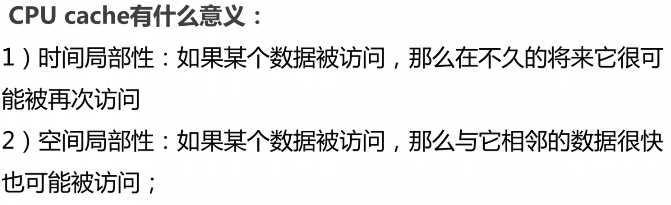


## 第2节并发基础

### 2.1CPU多级缓存--缓存一致性







·缓存一致性（MESI，Cache line的四种状态）

M：Modified，被修改

E：Exclusive，独享

S：Shared，共享

I：Invalid，无效

·CPU对缓存的四种操作可能会产生不一致的状态，因此缓存控制器监听到本地操作和远程操作，需要对地址一定的Cache line做出一定的修改。

## 第3节 项目准备

### 3.1环境初始化

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.21.RELEASE</**version**>  <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>  <**groupId**>com.byf</**groupId**>  <**artifactId**>concurrency</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**>  <**packaging**>war</**packaging**>  <**name**>concurrency</**name**>  <**description**>Demo project for Spring Boot</**description**>   <**properties**>  <**java.version**>1.8</**java.version**>  </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot</**artifactId**>  <**version**>1.5.21.RELEASE</**version**>  </**dependency**>  </**dependencies**>   <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**>  *<!-- 配置jar包的私服仓库 -->* <**repositories**>  <**repository**>  <**id**>app\_repository</**id**>  <**name**>app\_repository</**name**>  <**url**>http://127.0.0.1:8081/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  </**repository**>  </**repositories**>  *<!-- 配置插件的私服仓库 -->* <**pluginRepositories**>  <**pluginRepository**>  <**id**>app\_repository</**id**>  <**name**>app\_repository</**name**>  <**url**>http://127.0.0.1:8081/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  </**snapshots**>  </**pluginRepository**>  </**pluginRepositories**> </**project**> |

### 3.2案例准备

·github新建仓库concurrency

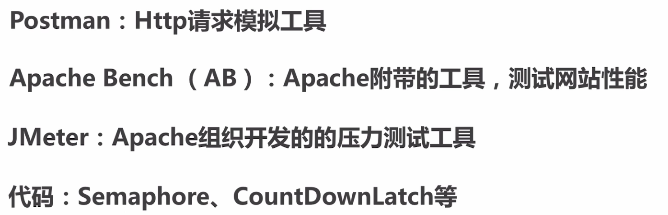
·本地在G:\concurrent\_programming下克隆项目

|  |
| --- |
| > git clone git@github.com:byf312358196/concurrency.git |

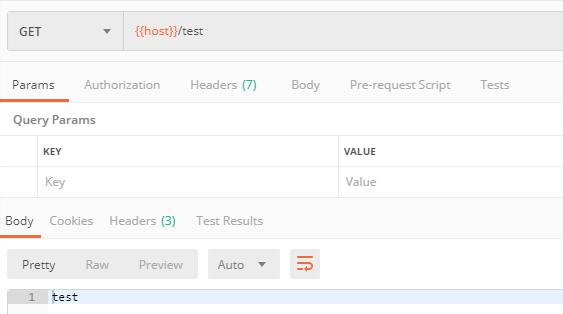
·start.spring.io新建springboot项目

·git提交代码

### 3.3并发模拟-工具



·postman



·ab

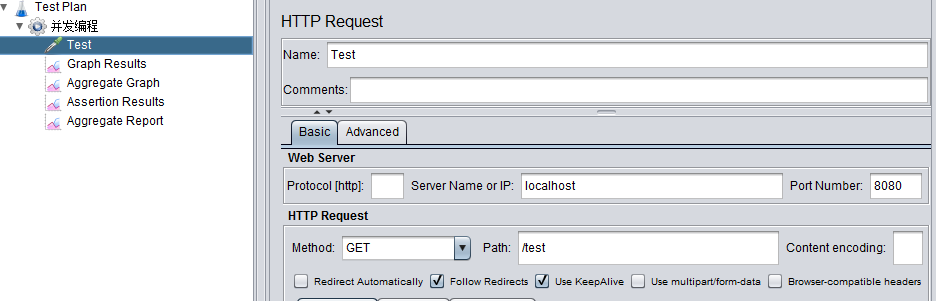
ab -n 100 -c 5 <http://localhost:8080/test>

·Jmeter

|  |
| --- |
| Error: Java version -- 1.7.0\_80 -- is too low to run JMeter. Needs a Java version greater than or eq |

Jmeter指定jdk版本

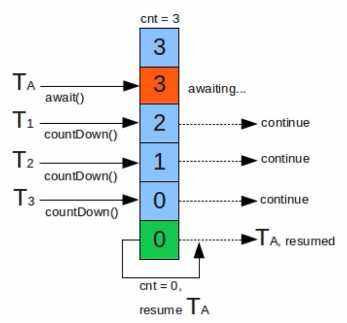
|  |
| --- |
| setlocal  set MINIMAL\_VERSION=1.8.0  set JAVA\_HOME=F:\Program Files\Java\jdk1.8.0\_172  set PATH=%JAVA\_HOME%\bin;%PATH% |



### 3.4并发模拟-代码

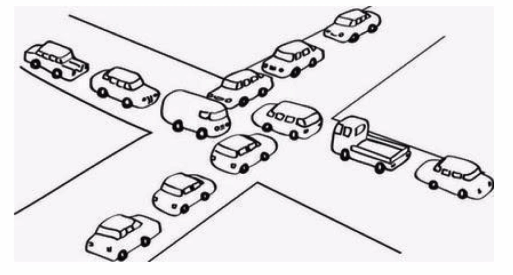
·CountDownLatch

等待countdown线程执行完，再进行其他处理



·Semaphore

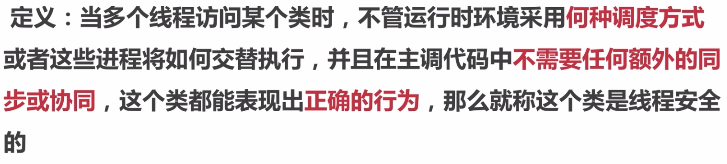
控制线程的并发数，同一时间仅有两个线程在抢占CPU，一个执行，一个阻塞，其他线程处于挂起，等待被唤醒



|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ConcurrencyTest {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;   **private static void** add(){  *count*++;  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |
| 10:04:01.783 [main] INFO com.byf.concurrency.counter.ConcurrencyTest - count:4963 |

## 第4节线程安全性

### 4.1原子性-AtomicLong、LongAdder



|  |
| --- |
| **private static** AtomicInteger *count* = **new** AtomicInteger(0);  **private static void** add(){  *count*.getAndIncrement(); } |
| **public final int** getAndAddInt(Object var1, **long** var2, **int** var4) {  **int** var5;  **do** {  // 从底层获取对象var1当前的值var2，赋给var5  var5 = **this**.getIntVolatile(var1, var2);  // 比较从对象var1取得当前值var2如果期待的是var5，就把var5加上var4，与预期不符，则重新从底层获取对象var1当前的值var2赋给var5，继续比较  } **while**(!**this**.compareAndSwapInt(var1, var2, var5, var5 + var4));   **return** var5; } |

|  |
| --- |
| **private static** LongAdder *count* = **new** LongAdder();  **private static void** add(){  *count*.increment(); } |
| **原理：LongAdder将不同线程通过Hash算法对数组中的Hash坐标对应的值进行增加，最后求和，分散了Atomic单点统计的压力。但并发更新可能导致统计的数据有些误差。**  **实际使用中，低并发使用Atomic优先，无需分cell求和计算；**  **高并发统计如果不是要求全局唯一，高精度的统计，优先使用LongAdder。** |

### 4.2原子性-AtomicReference、AtomicReferenceFieldUpdater

|  |
| --- |
| @Slf4j @ThreadSafe **public class** AtomicReferenceTest {  **private static** AtomicReference<Integer> *count* = **new** AtomicReference<>(0);   **public static void** main(String[] args) {  *count*.compareAndSet(0,2); *// 2  count*.compareAndSet(0,2); *// no  count*.compareAndSet(1,3); *// no  count*.compareAndSet(2,4); *// 4  count*.compareAndSet(3,5); *// no* ***log***.info(**"count:{}"**,*count*);  } } |
| 11:01:07.790 [main] INFO com.byf.concurrency.counter.atomic.AtomicReferenceTest - count:4 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** AtomicReferenceFieldUpdaterTest {   @Getter  **private volatile int count** = 100;  **private static** AtomicIntegerFieldUpdater<AtomicReferenceFieldUpdaterTest> *updater* =  AtomicIntegerFieldUpdater.*newUpdater*(AtomicReferenceFieldUpdaterTest.**class**, **"count"**);   **private static** AtomicReferenceFieldUpdaterTest *test* = **new** AtomicReferenceFieldUpdaterTest();  **public static void** main(String[] args) {  **if** (*updater*.compareAndSet(*test*,100,200)){  ***log***.info(**"count{}"**, *test*.getCount());  }   **if** (*updater*.compareAndSet(*test*,100,200)){  ***log***.info(**"update success count:{}"**, *test*.getCount());  } **else** {  ***log***.error(**"update failed count:{}"**, *test*.getCount());  }  } } |
| 11:00:13.660 [main] INFO com.byf.concurrency.counter.atomic.AtomicReferenceFieldUpdaterTest - count200  11:00:13.660 [main] ERROR com.byf.concurrency.counter.atomic.AtomicReferenceFieldUpdaterTest - update failed count:200 |

AtomicStampReference：CAS的ABA问题

线程1在访问共享数据是的值时，线程2将A的值改为B，很快又将B改回A，此时，线程1操作的值的版本其实已经发生了改变，这与设计思想不符。

线程1：我期望的值是A时，才进行修改；

线程2：我期望是A改成B，然后期望是B改成A；

线程1：比较与自己期望的值，发现此时是A，进行修改操作；

期望并发场景下，某段代码只执行一次

|  |
| --- |
| **private static** AtomicBoolean *isHappend* = **new** AtomicBoolean(**false**);  **private static void** test(){  **if** (*isHappend*.compareAndSet(**false**,**true**)){  ***log***.info(**"execute, isHaddped:{}"**, *isHappend*);  } } |
| **11:13:11.391 [pool-1-thread-1] INFO com.byf.concurrency.counter.atomic.ConcurrencyAtomicBooleanTest - execute, isHaddped:true**  **11:13:11.407 [main] INFO com.byf.concurrency.counter.atomic.ConcurrencyAtomicBooleanTest - isHapped:true** |

### 4.3原子性-Synchronized

1. 修饰代码块、修饰静态

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** SynchronizedTest1 {   *// 修饰代码块* **public void** test1(**int** j){  **synchronized** (**this**){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test1 {} -> {}"**,j, i);  }  }  }  *// 修饰方法* **public synchronized void** test2(**int** j){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test2 {} -> {}"**, j, i);  }  }   **public static void** main(String[] args) {  SynchronizedTest1 t1 = **new** SynchronizedTest1();  SynchronizedTest1 t2 = **new** SynchronizedTest1();  ExecutorService exec = Executors.*newCachedThreadPool*();  exec.execute(()->{  t1.test1(1);  });   exec.execute(()->{  t2.test2(2);  });  exec.shutdown();  }  } |

1. 修饰静态方法、使用类锁

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SynchronizedTest2 {   *// 修饰类* **public void** test1(**int** j){  **synchronized** (SynchronizedTest2.**class**){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test1 {} -> {}"**,j, i);  }  }  }  *// 修饰方法* **public synchronized static void** test2(**int** j){  **for** (**int** i=0;i<10;i++){  ***log***.info(**"test2 {} -> {}"**, j, i);  }  }   **public static void** main(String[] args) {  SynchronizedTest2 t1 = **new** SynchronizedTest2();  SynchronizedTest2 t2 = **new** SynchronizedTest2();  ExecutorService exec = Executors.*newCachedThreadPool*();  exec.execute(()->{  t1.test1(1);  });   exec.execute(()->{  t2.*test2*(2);  });  exec.shutdown();  }  } |

1. 修改计数器使成为线程安全

|  |
| --- |
| @Slf4j @ThreadSafe **public class** ConcurrencyTest3 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private static int** *count* = 0;   **private synchronized static void** add(){  *count*++;  }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

原子性--对比

·Synchronized：不可中断锁，适合竞争不激烈，可读性好。在并发高的场景下，性能下降，偏向锁-->轻量级锁-->自旋锁

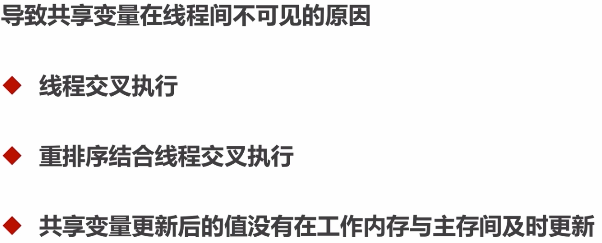
·Lock：可中断锁，多样化同步，竞争激烈能维持同步；

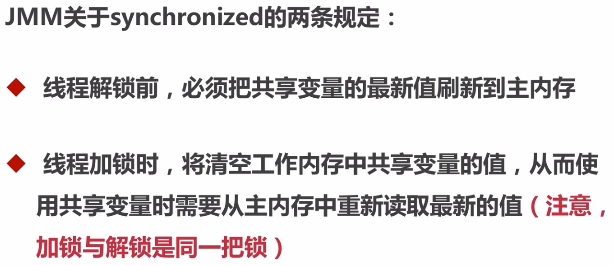
·Atomic：竞争激烈能维持常态，比Lock性能好（CAS，存在ABA问题）；只能维持同步一个值；

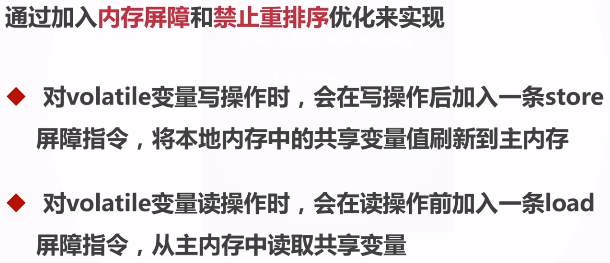
### 4.4可见性

1. 导致共享变量在线程间不可见的原因

·主内存与工作内存







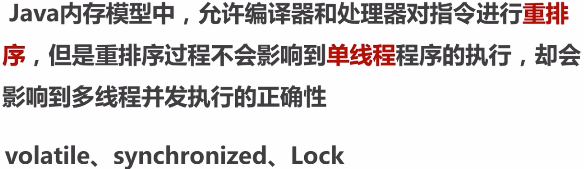




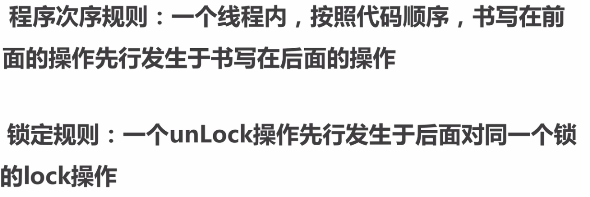
不适合做计数的修改型操作

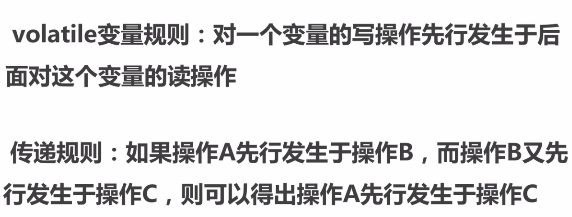
|  |
| --- |
| @Slf4j @NotThreadSafe **public class** ConcurrencyTest4 {  **private final static int *clientTotal*** = 5000;  **private final static int *threadTotal*** = 200;  **private volatile static int** *count* = 0;   **private static void** add(){  *count*++;  *// 1.读count值  // 2.count+1  // 3.写count值* }   **public static void** main(String[] args) **throws** InterruptedException {  ExecutorService exec = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(***threadTotal***);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(***clientTotal***);  **for** (**int** i=0; i<***clientTotal***;i++){  exec.execute(()->{  **try** {  semaphore.acquire();  *add*();  semaphore.release();  } **catch** (InterruptedException e) {  ***log***.error(**"exception"**,e);  }  countDownLatch.countDown();  });  }  countDownLatch.await();  exec.shutdown();  ***log***.info(**"count:{}"**,*count*);  } } |

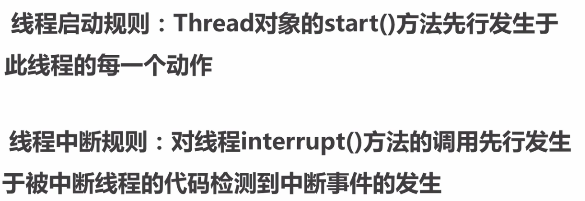
### 4.5有序性与总结

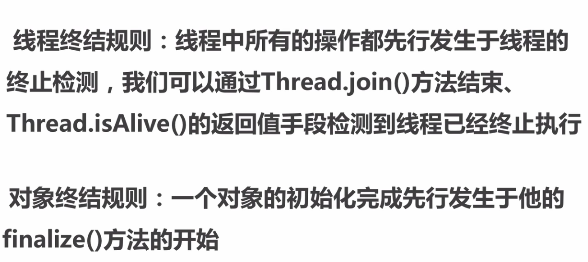


1. happens-before原则

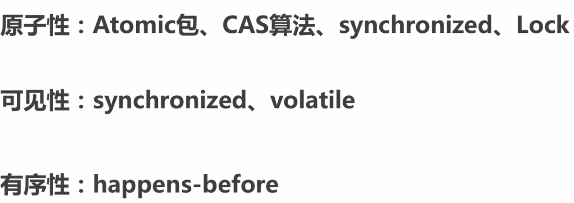






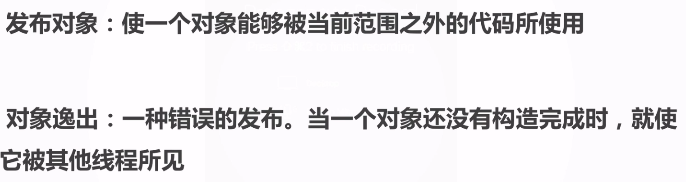


1. 总结



## 第5节安全发布对象

### 5.1发布与逸出



|  |
| --- |
| @Slf4j  @NotThreadSafe **public class** UnsafePublish {  **private** String[] **states** = {**"a"**, **"b"**, **"c"**};   **public** String[] getStates() {  **return states**;  }   **public static void** main(String[] args) {  UnsafePublish publish = **new** UnsafePublish();  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));   publish.getStates()[0] = **"d"**;  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));  } } |
| 17:34:20.254 [main] INFO com.byf.concurrency.publish.UnsafePublish - [a, b, c]  17:34:20.264 [main] INFO com.byf.concurrency.publish.UnsafePublish - [d, b, c] |

对象未构造完成前，不可以发布，防止新线程在构造期间，拿到发布的过期对象。

|  |
| --- |
| @Slf4j @NotThreadSafe **public class** UnsafePublish {  **private** String[] **states** = {**"a"**, **"b"**, **"c"**};   **public** String[] getStates() {  **return states**;  }   **public static void** main(String[] args) {  UnsafePublish publish = **new** UnsafePublish();  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));   publish.getStates()[0] = **"d"**;  ***log***.info(**"{}"**, Arrays.*toString*(publish.getStates()));  } } |
| 18:03:09.618 [main] INFO com.byf.concurrency.publish.UnsafePublish - [a, b, c]  18:03:09.618 [main] INFO com.byf.concurrency.publish.UnsafePublish - [d, b, c] |

|  |
| --- |
| @Slf4j @NotRecommend **public class** Escape {    **public** Escape(){  **new** InnerClass();  }   **private class** InnerClass{  **public** InnerClass(){  ***log***.info(**"{}"**, Escape.**this**.**thisCanBeEsape**);  }  }  **private int thisCanBeEsape** = 0;   **public static void** main(String[] args) {  **new** Escape();  } } |

### 5.2安全发布对象

单例的几种模式：

1. 不安全懒汉模式；
2. 安全饿汉模式（构造单例逻辑复制时，存在程序加载性能问题）；
3. 安全懒汉模式(Synchronized,并发高时存在性能问题)；
4. 双重检测枷锁懒汉单例模式（volatile禁止指令重排序）；

|  |
| --- |
| Instance = new Singleton();  // 1. memory = allocate() 分配对象的内存空间  // 2.ctorInstance() 初始化对象  // 3. instance = memory 设置instance指向刚分配的内存  如果上述步骤没有使用volatile，而指令重排序，步骤1和3执行后，步骤2还未执行，双重检测判断instance拿到的实例还未初始化构造，从而引入线程安全问题。 |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SingletonExample1 {  **private** SingletonExample1(){   }  **private volatile static** SingletonExample1 *instance* = **null**;   **public static** SingletonExample1 getInstance(){  **if** (*instance* == **null**){  **synchronized** (SingletonExample1.**class**){  **if** (*instance* == **null**){  *instance* = **new** SingletonExample1();  }  }  }  **return** *instance*;  }  **public static void** main(String[] args) {  SingletonExample1 singletonExample1 = **new** SingletonExample1();  }  } |

|  |
| --- |
| @Slf4j @ThreadSafe **public class** SingletonExample2 {  **private** SingletonExample2(){   }  *// 静态代码块的初始化顺序，从上往下；* **private static** SingletonExample2 *instance* = **null**;   **static** {  *instance* = **new** SingletonExample2();  }   **public static** SingletonExample2 getInstance(){  **return** *instance*;  }  **public static void** main(String[] args) {  System.***out***.println(*getInstance*().hashCode());  System.***out***.println(*getInstance*().hashCode());  } } |

### 5.3推荐枚举单例模式

|  |
| --- |
| @Slf4j @ThreadSafe @Recommend **public class** SingletonExample3 {  *// 私有构造函数* **private** SingletonExample3(){   }   **public static** SingletonExample3 getInstance(){  **return** Singleton.***INSTANCE***.getInstance();  }   **private enum** Singleton {  INSTANCE;  **private** SingletonExample3 singleton = **null**;  *// JVM保证这个方法绝对只调用一次* Singleton(){  singleton = **new** SingletonExample3();  }  **private** SingletonExample3 getInstance(){  **return** singleton;  }  }   **public static void** main(String[] args) {  System.***out***.println(SingletonExample3.*getInstance*());  } } |

