Оглавление

[Мотивация 3](#_Toc67997154)

[Закон Амдала 3](#_Toc67997155)

[Параллелизм в Java 3](#_Toc67997156)

[Проблемы параллельных программ 4](#_Toc67997157)

[java.lang.Thread 4](#_Toc67997158)

[Tread dump 4](#_Toc67997159)

[Создание потока 5](#_Toc67997160)

[Жизненный цикл потока 5](#_Toc67997161)

[Практика 6](#_Toc67997162)

[Метод run() 9](#_Toc67997163)

[Прерывание потока 9](#_Toc67997164)

[Практика 9](#_Toc67997165)

[Возможности встроенной синхронизации 12](#_Toc67997166)

[Ключевое слово synchronized 12](#_Toc67997167)

[Ключевое слово synchronized 13](#_Toc67997168)

[Ожидание и уведомление 13](#_Toc67997169)

[Практика 13](#_Toc67997170)

[Модель памяти 21](#_Toc67997171)

[Атомарные операции 21](#_Toc67997172)

[Видимость 22](#_Toc67997173)

[happens‑before 22](#_Toc67997174)

[Практика 23](#_Toc67997175)

[java.util.cuncurrent.atomic 27](#_Toc67997176)

[Практика 30](#_Toc67997177)

[Semaphore 33](#_Toc67997178)

[Практика 34](#_Toc67997179)

[CountDownLatch 37](#_Toc67997180)

[Практика 38](#_Toc67997181)

[CyclicBarrier 39](#_Toc67997182)

[ReentrantLock 39](#_Toc67997183)

[Condition 41](#_Toc67997184)

[Практика 41](#_Toc67997185)

[Источники 46](#_Toc67997186)

# Мотивация

Одновременное выполнение нескольких действий (например, отрисовка пользовательского интерфейса и передача файлов по сети)

Ускорение вычислений (при наличии нескольких вычислительных ядер)

# Закон Амдала

S — ускорение,

P — доля вычислений, которые возможно распараллелить,

N – количество вычислительных ядер.

# Параллелизм в Java

Запуск нескольких JVM на одной или на разных компьютерах (нет общей памяти, взаимодействие через файловую систему или сетевое соединение)

Запуск нескольких потоков внутри JVM (есть общая память, обширная поддержка в языке и стандартной библиотеке)

# Проблемы параллельных программ

Гонка (race condition). Ситуация, когда 2 потока пытаются работать с общей памятью (переменными), и один решает что‑то изменить в этих данных. Второй поток может этого не увидеть; может увидеть, но не все; может увидеть все, но не в том порядке. Каждая из этих ситуаций может привести к непредсказуемым последствиям. Для избегания нужно использовать примитивы синхронизации, которые дают эксклюзивный доступ (с каждыми типами данных может работать только один поток).

Взаимная блокировка (deadlock). 1‑й поток ждет 2‑й ресурс и занимает 1‑й ресурс, а 2‑й поток занимает 2‑й ресурс и ждет 1‑й.

# java.lang.Thread

Потоки представлены экземплярами класса java.lang.Thread. Методы: String getName(), long getId(), boolean isDeamon() (потоки фонового режима, при завершении), StackTraceElement[] getStackTrace(), ThreadGroup getThreadGroup().

# Tread dump

Список всех потоков с их состояниями и stack trace’ами: в консоли Ctrl+Break (windows) или Ctrl+\ (linux); jps -l, затем jstack PID; кнопка в IDE (Dump Threads, значок фотоаппарата)

# Создание потока

Подкласс Thread

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public class NewThread extends Thread {  @Override  public void run() {  // ...  }  } |
| 1.  2.  3.  4.  5. | public class Main {  public static void main(String[] args) {  new NewThread().start();  }  } |

Runnuble

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public class Main {  public static void main(String[] args) {  Runnable runnable = () -> ...;  new Thread(runnable).start();  }  } |

# Жизненный цикл потока

Создание объекта Thread

Запуск (thread.start())

Работа (выполнение метода run(), thread.isAlive() == true)

Завершение (завершение метода run() или исключение, нельзя перезапусить)

# Практика

Пример 1

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class Main1 {  public static void main(String[] args) {  for (int i = 0; i < 10; i++) {  Thread thread = new HellowThread();  thread.setName(i + "\_thread");  thread.start();  }  System.out.println("Hello from main\_thread");  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class HellowThread extends Thread {  @Override  public void run() {  System.out.println("Hello from " + getName());  }  } |

Не гарантируется порядок:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Hello from 1\_thread  Hello from 4\_thread  Hello from 2\_thread  Hello from 3\_thread  Hello from 0\_thread  Hello from 5\_thread  Hello from main\_thread  Hello from 6\_thread  Hello from 7\_thread  Hello from 8\_thread  Hello from 9\_thread |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Hello from 0\_thread  Hello from 5\_thread  Hello from 2\_thread  Hello from 7\_thread  Hello from 1\_thread  Hello from 9\_thread  Hello from 4\_thread  Hello from main\_thread  Hello from 3\_thread  Hello from 8\_thread  Hello from 6\_thread |

Пример 2

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public class Main2 {  public static void main() {  for (int i = 0; i < 10; i++) {  Thread thread =  new Thread(new HellowRunnable());  thread.setName(i + "\_thread");  thread.start();  }  System.out.println("Hello from main\_thread");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public class HellowRunnable implements Runnable {  @Override  public void run() {  System.out.println(  "Hello from "  + Thread.currentThread().getName());  }  } |

Пример 3

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public class Main3 {  public static void main() {  for (int i = 0; i < 10; i++) {  new Thread(  () -> System.out.println(  "Hello from"))  .start();  }  System.out.println("Hello from main\_thread");  }  } |

Пример 4

Можно так, но очень осторожно. Нужно понимание, если в этом объекте какое‑то состояние, и что будет если 10 потоков будут работать с этим состоянием. Здесь состояния нет, метод run() можно совершенно безопасно вызывать из многих потоков одновременно, ничему это не повредит (здесь только чтение из памяти, нет записи в память).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class Main4 {  public static void main() {  HellowRunnable hellowRunnable =  new HellowRunnable();  for (int i = 0; i < 10; i++) {  Thread thread = new Thread(hellowRunnable);  thread.setName(i + "\_thread");  thread.start();  }  System.out.println("Hello from main\_thread");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public class HellowRunnable implements Runnable {  @Override  public void run() {  System.out.println(  "Hello from "  + Thread.currentThread().getName());  }  } |

# Метод run()

Метод run() вместо того, чтобы стартовать новый поток, просто исполняет то, что ему передали в текущем потоке, то есть просто исолняет метод, а не создает новый поток.

# Прерывание потока

thread.interrupt()

Если поток находится в ожидании (sleep, join, wait), то ожидание прерывается исключением InterruptedException

Иначе у потока просто устанавливается флаг interrupted: флаг проверяется методами interrupted() и isInterrupted(); проверять флаг и завершать поток надо самостоятельно

thread.join() (ожидаем остановки interrupt(), если поток не останавливается, мы можем из него выйти и сделать что‑то еще)

# Практика

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class Main2\_1 {  public static void main() throws Exception {  Thread worker = new WorkerThread();  Thread sleeper = new SleeperThread();  System.out.println("Starting threads");  worker.start();  sleeper.start();  Thread.sleep(100L);  System.out.println("Interrupting threads");  worker.interrupt();  sleeper.interrupt();  System.out.println("Joining threads");  worker.join();  sleeper.join();  System.out.println("All done");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class SleeperThread extends Thread {  @Override  public void run() {  try {  Thread.sleep(10\_000L);  } catch (InterruptedException e) {  System.out.println("Sleep unterrupted");  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class WorkerThread extends Thread {  @Override  public void run() {  long sum = 0;  for (int i = 0; i < 1\_000\_000\_000; i++) {  sum += i;  if (i % 100 == 0 && isInterrupted()) {  System.out.println(  "Loop imterrupted at i = " + i +  ", summ = " + sum);  break;  }  }  }  } |

Результат

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Starting threads  Interrupting threads  Joining threads  Loop imterrupted at i = 27718400, summ = 384154863139200  Sleep unterrupted  All done |

Если убрать interrupt() и join(), то программа выполниться и повиснет, т. к. будут выполняться потоки worker и sleeper

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | public class Main2\_2 {  public static void main() throws Exception {  Thread worker = new WorkerThread();  Thread sleeper = new SleeperThread();  System.out.println("Starting threads");  worker.start();  sleeper.start();  Thread.sleep(100L);  System.out.println("All done");  }  } |

Если сделать потоки worker и sleeper демонами, то программа завершиться моментально.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | public class Main2\_3 {  public static void main() throws Exception {  Thread worker = new WorkerThread();  worker.setDaemon(true);  Thread sleeper = new SleeperThread();  sleeper.setDaemon(true);  System.out.println("Starting threads");  worker.start();  sleeper.start();  Thread.sleep(100L);  System.out.println("All done");  }  } |

# Возможности встроенной синхронизации

Взаимное исключение (пока один поток что‑то делает, другие не могут ему помешать)

Ожидание уведомления (поток ожидает уведомлений от других потоков)

# Ключевое слово synchronized

Синхронизованный метод

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public synchronized void doSomething {  // ...  } |

Синхронизованный блок внутри метода

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | public void doSomething {  synchronized (obj) {  // ...  }  } |

(obj) = this или (obj) = Class.class (статический метод)

# Ключевое слово synchronized

Синхронизация блоков — по монитору указанного объекта

Синхронизация методов — по монитору текущего объекта (this)

Синхронизация статических методов — по монитору класса

# Ожидание и уведомление

Допустимы только внутри synchronized

void wait(), void wait(long millis), void wait(long millis, int nanos)

void notify(), void notyfyAll()

# Практика

Пример 1

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class Main3\_1 {  public static void main() throws Exception {  Account account = new Account(100\_000);  System.out.println(  "Begin balance = "  + account.getBalance());  Thread withdrawThread =  new WithdrawThread(account);  Thread depositThread =  new DepositThread(account);  withdrawThread.start();  depositThread.start();  withdrawThread.join();  depositThread.join();  System.out.println(  "End balance = " + account.getBalance());  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37. | public class Account {  private long balance;  public Account() {  this(0L);  }  public Account(long balance) {  this.balance = balance;  }  public long getBalance() {  return balance;  }  public void deposit(long amount) {  checkAmountNonNegative(amount);  balance += amount;  }  public void withdraw(long amount) {  checkAmountNonNegative(amount);  if (balance < amount) {  throw new IllegalArgumentException(  "not enough money");  }  balance -= amount;  }  private static void checkAmountNonNegative(  long amount) {  if (amount < 0) {  throw new IllegalArgumentException(  "negative amount");  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class DepositThread extends Thread {  private final Account account;  public DepositThread(Account account) {  this.account = account;  }  @Override  public void run() {  for (int i = 0; i < 20\_000; i++) {  account.deposit(1);  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class WithdrawThread extends Thread {  private final Account account;  public WithdrawThread(Account account) {  this.account = account;  }  @Override  public void run() {  for (int i = 0; i < 20\_000; i++) {  account.withdraw(1);  }  }  } |

В задаче мы 20\_000 раз снимаем и кладем деньги, но по итогу получаем баланс, который отличается от изачального. Мы попадаем на дата рейс (состояние гонки, race condition). Причина в том, что операции инкремент и декремент не атомарны, т. е. один из потоков может положить результат своей работы поверх другого.

|  |  |
| --- | --- |
| 1.  2. | Begin balance = 100000  End balance = 92534 |
| 1.  2. | Begin balance = 100000  End balance = 101586 |
| 1.  2. | Begin balance = 100000  End balance = 87135 |

Пример 2

Способ решения при помощи ключевого слова synchronized над методами изменения счета, т. е. мы говорим, что в один момент времени два потока не могут заходить в эти методы, эти два метода в каждый момент времени выполняться только в одном потоке. Не может такого быть, что мы вошли в метод deposit() из одного потока и из другого потока и это один и тот же объект. Кроме того не может быть такого, что мы в метод deposite() зашли из одного потока и на том же объекте из другого потока в метод withdraw() зашли, потому что мы синхронизируемся по данному экземпляру, но а он общий (если у 2‑х потоков одно и та же ссылка).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38. | public class Account {  private long balance;  public Account() {  this(0L);  }  public Account(long balance) {  this.balance = balance;  }  public long getBalance() {  return balance;  }  public synchronized void deposit(long amount) {  checkAmountNonNegative(amount);  balance += amount;  }  public synchronized void withdraw(  long amount) {  checkAmountNonNegative(amount);  if (balance < amount) {  throw new IllegalArgumentException(  "not enough money");  }  balance -= amount;  }  private static void checkAmountNonNegative(  long amount) {  if (amount < 0) {  throw new IllegalArgumentException(  "negative amount");  }  }  } |

Пример 3

Рекомендуется в synchronized запихивать только какой‑то маленький кусочек работы, который реально не терпит многопоточного доступа. В данном случае вся полезная работа оказывается в synchronized блоке и никакого прироста производительности у нас тут не будет. Но в большинстве реальных ситуаций можно выделить маленький кусочек, который требует синхронизации. Мы маленький кусочек работы программы действительно синхронизируем, обеспечиваем там, чтобы состояние памяти было консистентно, а другие операции из под этой синхронизации можно вынести, они исполняются по настоящему параллельно.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41. | public class Account {  private long balance;  public Account() {  this(0L);  }  public Account(long balance) {  this.balance = balance;  }  public long getBalance() {  return balance;  }  public void deposit(long amount) {  checkAmountNonNegative(amount);  synchronized (this) {  balance += amount;  }  }  public void withdraw(long amount) {  checkAmountNonNegative(amount);  synchronized (this) {  if (balance < amount) {  throw new IllegalArgumentException(  "not enough money");  }  balance -= amount;  }  }  private static void checkAmountNonNegative(  long amount) {  if (amount < 0) {  throw new IllegalArgumentException(  "negative amount");  }  }  } |

Пример 4

Пример с ожиданием пополнения. Появился еще один метод waitAndWithdraw(), который включает в себя ожидание. Это synchronized метод, поэтому внутри него можно вызывать wait() на текущем объект (на this). Здесь мы в цикле проверяем а правда ли, что мы можем нужную операцию выполнить, т. е. достаточно ли на счету денег. Если не достаточно, то отправляем поток в спячку до тех пор, пока его кто‑нибудь не пробудет своим notify(). Пока этот поток спит данный монитор может быть захвачен каким‑нибудь другим потоком. Поэтому метод deposit(): вызывает notifyAll(). notifyAll() рассылает уведомления всем потокам, которые спят на том самом мониторе, на котором мы здесь синхронизировались, т. е. тот же самый экземпляр класса Account (this). Каждый раз, когда выполняется пополнение счета, после того как поток освобождает метод deposite() поток, ждущий в методе waitAndWithdraw(), проверяет условие. Если все хорошо, то снимает деньги и завершается, если денег не достаточно, то опять отправляется в спячку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class Main3\_4 {  public static void main() throws Exception {  Account account = new Account(0);  new DepositThread(account).start();  System.out.println(  "Calling waitAndWithdraw()...");  account.waitAndWithdraw(50\_000\_000);  System.out.println(  "waitAndWithdraw() finished");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40. | public class Account {  private long balance;  public Account() {  this(0L);  }  public Account(long balance) {  this.balance = balance;  }  public long getBalance() {  return balance;  }  public synchronized void deposit(long amount) {  checkAmountNonNegative(amount);  balance += amount;  notifyAll();  }  public synchronized void waitAndWithdraw(  long amount)  throws InterruptedException {  checkAmountNonNegative(amount);  while (balance < amount) {  wait();  // System.out.println("Wakeup: " + balance);  }  balance -= amount;  }  private static void checkAmountNonNegative(  long amount) {  if (amount < 0) {  throw new IllegalArgumentException(  "negative amount");  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class DepositThread extends Thread {  private final Account account;  public DepositThread(Account account) {  this.account = account;  }  @Override  public void run() {  for (int i = 0; i < 50\_000\_000; i++) {  account.deposit(1);  }  }  } |

# Модель памяти

Модель памяти — спецификация того, как будут работать ваши программы, в случае, если запущенно несколько потоков. Более конкретно: какие операции являются атомарными, какие нет (17 глава Java language спецификейшн). Второе важное свойство — видимость, когда состояние сделанные одним потоком видны в другом.

# Атомарные операции

Чтение и запись полей всех типов, кроме long и double, происходит атомарно

Если поле объявлено с модификатором volatile, то атомарно читаются и пишутся даже long и double

# Видимость

Изменения значений полей, сделанные одним потоком, могут быть не видны в другом потоке

Изменения, сделанные одним потоком, могут быть видны в другом потоке в ином порядке

Правила формализованы при помощи отношения happens‑before (отношение: если в одном потоке произошло некое событие x, а в другом потоке произошло событие z, то мы гарантировано знаем, что все произошло до x, будет видно после z; вопрос в том, что это за пары таких событий, и как их нами обеспечить)

Семантика final

# happens‑before

Запись volatile‑поля happens‑before чтения этого поля

Освобождение монитора happens‑before захват того же монитора

thread.start() happens‑before thread.run()

Завершение thread.run happens‑before выход из thread.join()

# Практика

Пример 1

В многопоточной ситуации эта программа не корректна, потому что метод getInstance() не синхронизованный (одновременной зайдут несколько потоков и создадут объект).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | public class Singleton1 {  private int foo;  private String bar;  private Singleton1() {  this.foo = 13;  this.bar = "zap";  }  private static Singleton1 instance;  public static Singleton1 getInstance() {  if (instance == null) {  instance = new Singleton1();  }  return instance;  }  } |

Пример 2

Способ решения проблемы при помощи ключевого слова synchronized. Некоторым здесь не нравиться, что приходиться платить накладные расходы за синхронизацию на каждый вызов getInstance(), в то время как реально хотелось бы заплатить только первый раз, когда реально происходит инициализация.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public class Singleton2 {  private int foo;  private String bar;  private Singleton2() {  this.foo = 13;  this.bar = "zap";  }  private static Singleton2 instance;  public synchronized static  Singleton2 getInstance() {  if (instance == null) {  instance = new Singleton2();  }  return instance;  }  } |

Поэтому некоторые светлые умы начинают изобретать такую конструкцию:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class Singleton3 {  private int foo;  private String bar;  private Singleton3() {  this.foo = 13;  this.bar = "zap";  }  private static Singleton3 instance;  public synchronized static  Singleton3 getInstance() {  if (instance == null) {  synchronized (Singleton3.class) {  if (instance == null) {  instance = new Singleton3();  }  }  }  return instance;  }  } |

Т. е. если instance создан, то мы не пойдем в instance блок, не будем платить за синхронизацию. Это идиома даже получила специальное название ‑ антипаттерн Double checked locking (двойная проверка и лок). Она не корректна. На вопрос почему может помочь знание о модели памяти Java. В частности нужно понимать, что инициализация (instance = new Singleton3()) это не только присвоение ссылки на новый объект (new Singleton3()) в поле instance, но это еще присвоение значений полей, создаваемого экземпляра. Эти записи присвоения полей экземпляра (в конструкторе) и присвоение этой статической переменной (17) они могут быть произвольным образом переупорядочены. Мы уже записали в это поле instance ссылку на новый объект, при этом его значения полей еще не инициализированы. И получится, что ссылка на недостроенный объект утекла в нашу программу. Кто‑то на нем может вызвать какие‑то методы и начать получать странные исключения из‑за того, что состояние объекта некорректно.

Поэтому этот случай ошибочный, за исключение такого случая:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class Singleton4 {  private int foo;  private String bar;  private Singleton4() {  this.foo = 13;  this.bar = "zap";  }  private static volatile Singleton4 instance;  public synchronized static  Singleton4 getInstance() {  if (instance == null) {  synchronized (Singleton4.class) {  if (instance == null) {  instance = new Singleton4();  }  }  }  return instance;  }  } |

volatile нам обеспечивает happens‑before между записью в volatile ссылки на объект (17). Мы знаем, что присвоение полей должны были произойти до нее, до записи этой ссылки в объект. А в (14) мы делаем чтение из volatile переменной, соответственно гарантировано видим все то, что в памяти происходило в (17).

В лучшем случаем используйте второй способ (volatile метод).

# java.util.cuncurrent.atomic

AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference<V>

Операции: V get(), viod set(V newValue), boolean compareAndSet(V expect, V update)

Каждая из них (1) — всего лишь обертка над примитивом или дженерик. Зачем нужны такие обертки? Есть же volatile — гарантирует атомарность доступа на чтение и на запись. На самом деле они добавляют дополнительную атомарную операцию compareAndSet(), т .е гарантировано атомарно происходит следующее: мы сравниваем, что в данный момент внутри этого класса обертки лежит такое значение, и если это правда, то атомарно меняем его на другое. Это происходит атомарно, т. е. никто не может вклиниться в эту последовательность и нарушить ее. Это решение и в частности проблемы с тем, что инкремент или декремент это не атомарные операции обычных примитивных типов даже с volatile переменной (чтение и запись в нее будут атомарны, а инкремент — нет, потому что это сначала чтение, потом изменение и потом запись — операция не атомарна, может вклиниться другой поток).

Примитив compareAndSet() позволяет реализовать другие операции

Пример из AtomicInteger:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31. | public class AtomicInteger {  ...  public final int get() {  ...;  }  ...  public final boolean compareAndSet(  int expect,  int update) {  ...  }  ...  public final int incrementAndGet() {  for (;;) {  int current = get();  int next = current + 1;  if (compareAndSet(current, next)) {  return next;  }  }  }  ...  } |

Додуматься как реализовать getAndIncrement() (разница как префиксный и постфиксный инкремент).

Помимо представления других операций compareAndSet() хороша еще тем, что в современных процессорах есть такая команда, машинная программа compareAndSet(), которая ровно это и делает. Т. е. получается реализация достаточно дешевой не надо чего‑то лочить, обвязывать сложными примитивами синхронизации, на машинном уровне compareAndSet() будет одной командой.

Оборачиваю в атомарные обертки отдельные переменные мы можем достичь атомарности работы с каждой отдельной переменной, но мы не можем решить задачу, например, когда нам нужно консистентно обновить значение 2‑х и более переменных. Например, добавляем в ArrayList новые элементы. Нам нужно помимо того, чтобы в массив записать этот добавленный объект, нам нужно еще увеличить счетчик, который показывает сколько в массиве уже значений записано. Через такие атомарные переменные эту задачу не решить. Мы можем гарантировать, что атомарно обновиться переменная с текущим размером, но ничего не будем знать о том, как она соотносится с операцией записи значения в массив. Для этого все еще нужны synchronized блоки (методы) или один из других примитивов синхронизации (в следующем разделе).

# Практика

Пример 1

Если здесь не будет volatile, то не будет гарантии, что другие потоки увидят то, что сделали остальные. Но на самом деле даже это не поможет, потому что (3) не правильное решение задачи. Оно обеспечит атомарное чтение и запись, но не атомарный инкремент. Получится, что несколько потоков одно и то же значение получить. А у нас задача раздать потокам последовательность от 0 до N без повторений. Т. е. много потоков могут вызывать метод nextInt(), нужно чтобы каждый из них получал свое уникальное значение. Почему это не работает. Я уже много раз говорил, что инкремент это операция не атомарная, и сейчас я это продемонстрирую.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | public class Main5\_1 {  private static volatile int cointer = 0;  public static int nextInt() {  return cointer++;  }  public static void main() throws Exception {  List<Thread> threads = new ArrayList<>();  for (int i = 0; i < 10; i++) {  Thread thread = new Thread(() -> {  for (int j = 0; j < 1000; j++) {  nextInt();  }  });  thread.start();  threads.add(thread);  }  for (Thread thread : threads) {  thread.join();  }  System.out.println(  "Counter final value: " + cointer);  }  } |

В этой программе я запускаю 10 потоков, каждый из них по 1000 раз делает nextInt(). И в конце печатаем финальное значение переменной счетчика и по нему делаем заключение, каждый ли поток получил свое уникальное значение или не каждый:

|  |  |
| --- | --- |
| 1. | Counter final value: 10000 |
| 1. | Counter final value: 10000 |
| 1. | Counter final value: 10000 |
| 1. | Counter final value: 9606 |

Ошибку удалось поймать на 4 раз. В этом коварство многопоточных программ (могла бы выйти на 1001 раз).

Пример 2

Правильное решение:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | public class Main5\_2 {  private static int cointer = 0;  public synchronized static int nextInt() {  return cointer++;  }  public static void main() throws Exception {  List<Thread> threads = new ArrayList<>();  for (int i = 0; i < 10; i++) {  Thread thread = new Thread(() -> {  for (int j = 0; j < 1000; j++) {  nextInt();  }  });  thread.start();  threads.add(thread);  }  for (Thread thread : threads) {  thread.join();  }  System.out.println(  "Counter final value: " + cointer);  }  } |

Сделать метод synchronized. Делать переменную volatile не обязательно.

Пример 3

Или использовать AtomicInteger:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31. | public class Main5\_2 {  // в конструкторе 0 по умолчанию  private static final AtomicInteger cointer =  new AtomicInteger();  public static int nextInt() {  return cointer.getAndIncrement();  }  public static void main() throws Exception {  List<Thread> threads = new ArrayList<>();  for (int i = 0; i < 10; i++) {  Thread thread = new Thread(() -> {  for (int j = 0; j < 1000; j++) {  nextInt();  }  });  thread.start();  threads.add(thread);  }  for (Thread thread : threads) {  thread.join();  }  System.out.println(  "Counter final value: " + cointer);  }  } |

# Semaphore

Класс java.util.cuncurrent.Semaphore

Ограничивает одновременно доступ к ресурсу

В отличии от synchronized‑блока, одновременно могут работать несколько потоков (но не более заданного N)

Операции void acquire(), void release()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Semaphore semaphore = new Semaphore(10);  try {  semaphore.acquire();  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  // ...  } finally {  semaphore.release();  } |

# Практика

Пример 1

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public class Main6\_1 {  public static void main() throws Exception {  Semaphore semaphore = new Semaphore(2);  List<Thread> threads = new ArrayList<>();  for (int i = 0; i < 10; i++) {  DemoThread thread =  new DemoThread(semaphore);  threads.add(thread);  thread.start();  }  Thread.sleep(20\_000);  for (Thread thread : threads) {  thread.interrupt();  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36. | import java.util.concurrent.Semaphore;  public class DemoThread extends Thread {  private final Semaphore semaphore;  public DemoThread(Semaphore semaphore) {  this.semaphore = semaphore;  }  @Override  public void run() {  try {  runUnsafe();  } catch (InterruptedException e) {  System.out.println(  getName() + " interrupted");  }  }  private void runUnsafe()  throws InterruptedException {  for (;;) {  semaphore.acquire();  try {  System.out.println(  getName() + " acquired semaphore");  Thread.sleep(5\_000L);  } finally {  System.out.println(  getName() + " releasing semaphore");  semaphore.release();  }  }  }  } |

Результат

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26. | Thread-2 acquired semaphore  Thread-1 acquired semaphore  Thread-2 releasing semaphore  Thread-1 releasing semaphore  Thread-1 acquired semaphore  Thread-2 acquired semaphore  Thread-1 releasing semaphore  Thread-1 acquired semaphore  Thread-2 releasing semaphore  Thread-2 acquired semaphore  Thread-2 releasing semaphore  Thread-2 acquired semaphore  Thread-1 releasing semaphore  Thread-1 acquired semaphore  Thread-2 releasing semaphore  Thread-8 interrupted  Thread-3 interrupted  Thread-2 interrupted  Thread-7 interrupted  Thread-6 interrupted  Thread-4 interrupted  Thread-10 interrupted  Thread-1 releasing semaphore  Thread-1 interrupted  Thread-5 interrupted  Thread-9 interrupted |

Пример 2

По динамике мы можем видеть, что в каждый момент времени Semaphore захвачен только 2‑мя потоками. Но при этом интересный спецэффект, что захватывают этот Semaphore все время одни и те же два потока (1‑2). Это проблема, что доступ распределяется несправедливо. Есть способ этого избежать и сказать, что треды на этом семафоре будут выстраиваться в честную очередь и получать доступ в порядке этой очереди. Для этого нам всего лишь нужно передать в конструктор «волшебное слово» (new Semaphore(2, true)):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public class Main6\_1 {  public static void main() throws Exception {  Semaphore semaphore = new Semaphore(2, true);  List<Thread> threads = new ArrayList<>();  for (int i = 0; i < 10; i++) {  DemoThread thread =  new DemoThread(semaphore);  threads.add(thread);  thread.start();  }  Thread.sleep(20\_000);  for (Thread thread : threads) {  thread.interrupt();  }  }  } |

В synchronized методах (блоках) та же самая проблема: никак не специфицирован порядок, в котором ждущие потоки будут попадать в synchronized блок. Может так получиться, что один и тот же поток все время будет попадать в этот synchronized блок и делать там свою работу, а всем остальным придется ждать. При этом для этой встроенной синхронизации synchronized методов нет способа обойти это ограничение, т. е. попросить потоки выстраиваться в очередь, а для библиотечных примитивов есть такая возможность. Это более мощный механизм, но и бывает, что более дорогой за счет поддержании этой очереди.

# CountDownLatch

Класс java.util.cuncurrent.CountDownLatch

Обеспечивает точку синхронизации между N потоками (несколько потоков могут дожидаться друг друга и потом стартовать одновременно)

Операции: void await(), void countDown() (уменьшает счетчик, переданный в конструктор)

# Практика

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class Main7 {  public static void main()  throws InterruptedException {  CountDownLatch latch = new CountDownLatch(10);  for (int i = 0; i < 10; i++) {  new DemoThread(latch).start();  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36. | public class DemoThread extends Thread {  private final CountDownLatch latch;  public DemoThread(CountDownLatch latch) {  this.latch = latch;  }  @Override  public void run() {  try {  runUnsafe();  } catch (InterruptedException e) {  System.out.println(  getName() + " interrupted");  }  }  private void runUnsafe()  throws InterruptedException {  Thread.sleep(  (long) (Math.random() \* 10\_000L));  System.out.println(  getName() + " finished initialization");  latch.countDown();  latch.await();  System.out.println(  getName() + " entered main phase");  Thread.sleep(  (long) (Math.random() \* 10\_000L));  }  } |

Результат

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | Thread-5 finished initialization  Thread-4 finished initialization  Thread-9 finished initialization  Thread-8 finished initialization  Thread-7 finished initialization  Thread-3 finished initialization  Thread-1 finished initialization  Thread-2 finished initialization  Thread-10 finished initialization  Thread-6 finished initialization  Thread-6 entered main phase  Thread-5 entered main phase  Thread-4 entered main phase  Thread-9 entered main phase  Thread-8 entered main phase  Thread-7 entered main phase  Thread-3 entered main phase  Thread-1 entered main phase  Thread-10 entered main phase  Thread-2 entered main phase |

# CyclicBarrier

Класс java.util.cuncurrent.CyclicBarrier

Вариант CountDownLatch, допускающий повторное ожидание

# ReentrantLock

Класс java.util.cuncurrent.locks.ReentrantLock

Обеспечивает взаимное исключение потоков, аналогичное synchronized‑блокам

Операции: lock(), unlock()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Lock lock = new ReentrantLock();  lock.lock();  try {  // ...  } finally {  lock.unlock();  } |

В отличии от synchronized мы можем захватить лок в одном методу и отпустить в другом. Кроме того у ReentrantLock есть расширенные возможности по организации ожидания (wait(), notify()). Есть похожий механизм, но чуть более мощный.

Также есть возможность передать честную очередь:

|  |  |
| --- | --- |
| 1. | Lock lock = new ReentrantLock(true); |

# Condition

Класс java.util.cuncurrent.locks.Condition

Аналог wait()/notify() с другими названиями

Привязан к Lock’у

У одного Lock’а может быть много Condition’ов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Lock lock = new ReentrantLock();  Condition condition = lock.newCondition();  lock.lock();  try {  while (true) {  condition.await();  }  } finally {  lock.unlock();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Lock lock = new ReentrantLock();  Condition condition = lock.newCondition();  lock.lock();  try {  condition.signal();  } finally {  lock.unlock();  } |

# Практика

Пример 1

Рассмотрим пример, который мы раньше рассматривали через встроенную синхронизацию, а в этот раз сделаем ее через Lock и Condition.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class Main8 {  public static void main() throws Exception {  Account account = new Account(0);  new DepositThread(account).start();  System.out.println(  "Entering waitAndWithdraw");  account.waitAndWithdraw(50\_000\_000);  System.out.println(  "waitAndWithdraw finished, end balance = "  + account.getBalance());  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | public class DepositThread extends Thread {  private final Account account;  public DepositThread(Account account) {  this.account = account;  }  @Override  public void run() {  for (int i = 0; i < 60\_000\_000; i++) {  account.deposit(1);  }  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59. | public class Account {  private final Lock lock = new ReentrantLock();  private final Condition balanceIncreased =  lock.newCondition();  private long balance;  public Account() {  this(0L);  }  public Account(long balance) {  this.balance = balance;  }  public long getBalance() {  lock.lock();  try {  return balance;  } finally {  lock.unlock();  }  }  public void deposit(long amount) {  checkAmountNonNegative(amount);  lock.lock();  try {  balance += amount;  balanceIncreased.signalAll();  } finally {  lock.unlock();  }  }  public void waitAndWithdraw(long amount)  throws InterruptedException {  checkAmountNonNegative(amount);  lock.lock();  try {  while (balance < amount) {  balanceIncreased.await();  // System.out.println("awake");  }  balance -= amount;  } finally {  lock.unlock();  }  }  private static void checkAmountNonNegative(  long amount) {  if (amount < 0) {  throw new IllegalArgumentException(  "negative amount");  }  }  } |

# ReentrantReadWriteLock

Класс java.util.cuncurrent.locks  
.ReentrantReadWriteLock

Поддерживает разделение доступа на чтение и на запись

Это лог с разделением на читать и писать. Идея в том, что когда у нас есть некая структура в памяти, то расшаренная между несколькими потоками, то потоки читающие эту структуру никак друг другу не мешают, т. е. мы можем допускать много потоков до чтения нашей обшей памяти. Они никак друг другу не помешают, ничего не испортят. Проблемы начинаются только когда мы начинаем эту структуру изменять. И вот тогда нужно прогнать все читающие потоки, им какое‑то время нельзя будет читать, чтобы не получить неконсистентность состояния, и убедиться что обновлять нашу общую память будет только один поток. После того, как он отработал, опять может либо получить доступ либо следующий писатель, либо все читатели сразу вернуться.

Это хорошо в случаях, когда у нас общая память часто читается и редко модифицируется. В таком случае параллелизм нашей программы будет больше, потому что читатели смогут одновременно работать и читать эту общую память. А изредко приходящие писатели получат свою блокировку на этот участок памяти, быстренько отработают и дальше программа будет работать полностью параллельно.

ReentrantLock будет лочить на любом доступе. Программа из многопоточной превращается в «Тыкву».

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | ReadWriteLock lock = new ReentrantReadWriteLock();  lock.readLock().lock();  try {  // ...  } finally {  lock.readLock().unlock();  }  lock.writeLock().lock();  try {  // ...  } finally {  lock.writeLock().unlock();  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40. |  |

# Источники

<https://www.youtube.com/watch?v=zxZ0BXlTys0&ab_channel=ComputerScienceCenter>

<https://www.youtube.com/watch?v=umTVNoG3760&t=792s&ab_channel=ComputerScienceCenter>