ГЛАВА 1

Введение в конкурентность

> Конкурентность — это когда удерживаешь в поле зрения множество задач одновременно.
>
> Параллелизм — это когда выполняешь множество задач одновременно.
>
> *Роб Пайк (Rob Pike)*

Чтобы по-настоящему оценить какое-либо явление, важно знать, как оно возникло, тем более если есть возможность отследить пройденные этапы и преодоленные трудности. Такой подход не только выдвигает на первый план текущие достижения, но и помогает осознать их значимость. Конкурентность в Java с момента своего появления прошла долгий путь. Потребовалось немало времени, чтобы она эволюционировала до теперешнего состояния. Прежде чем разбираться в современных достижениях — таких как виртуальные потоки и структурированная конкурентность, — необходимо проследить их появление и развитие. В этой главе я дам общее представление о конкурентности в Java, и мы кратко обсудим ее развитие.

Краткая история потоков в Java

Java изначально разрабатывали с прицелом на конкурентность. Это один из первых языков со встроенной поддержкой многопоточности. Год за годом возможности конкурентности в Java росли и совершенствовались. На этом пути были как успехи, так и ошибки, на которых приходилось учиться.

Развитие конкурентности в Java началось с базовой синхронизации и управления потоками. Затем в Java 5 добавили пакет java.util.concurrent (https://oreil.ly/5e8s1) с новыми возможностями, в том числе фреймворком Executor (https://oreil.ly/lDFVL), механизмом блокировок и конкурентными коллекциями. С появлением в Java 7 фреймворка Fork/Join многоядерные процессоры

получили способность конкурентно обрабатывать задачи. И уже совсем недавно, благодаря Project Loom (https://oreil.ly/gcaYC), преодолены сложность и ограничения традиционных потоков.

Проект предложил структурированную конкурентность и легкие потоки, построенные в пользовательском пространстве, без участия ядра ОС (user-mode threads). Все это призвано в конечном счете упростить и ускорить разработку конкурентного кода.

Почему так важны эти изменения? История эволюции конкурентности в Java — это неустанный поиск эффективности и простоты в условиях постоянно растущей сложности. Такая тенденция прослеживается не только в Java: она отражает как неизменные устремления сообщества разработчиков, так и траекторию развития индустрии ПО в целом.

Рассмотрим подробнее этапы развития конкурентности в Java и оценим достигнутый прогресс.

Язык Java построен на основе потоков

При проектировании языка Java конкурентность была явно обозначенной целью. Изначально встроенная поддержка потоков (threads) и концепция многопоточности — ключевые отличительные особенности данного языка. Благодаря этому Java-разработчики могут использовать конкурентность без оглядки на особенности конкретных операционных систем.

Поток в Java — это наименьшая единица выполнения. Он представляет собой независимый путь выполнения задачи внутри программы. Потоки разделяют одно адресное пространство, то есть имеют доступ к одним и тем же переменным, а также структурам данных программы. При этом у каждого потока есть собственный программный счетчик (program counter), стек и локальные переменные, что позволяет ему работать независимо. При необходимости такая архитектура упрощает взаимодействие между потоками.

Однако модель многопоточности в Java зависит от базовой операционной системы, которая планирует и выполняет потоки. Чтобы обеспечить эффективное выполнение, ОС распределяет процессорное время между потоками, управляя переходами между их состояниями. Распределяя потоки между несколькими процессорами (CPU), система достигает истинного параллелизма, повышая производительность конкурентных приложений (рис. 1.1).

Чем больше потоков в нашей Java-программе, тем больше сред выполнения мы фактически создаем. Мы получаем возможность выполнять несколько операций одновременно. Наиболее выгодно это для приложений, требующих высокого уровня параллелизма или конкурентности: для веб-серверов, пайплайнов

обработки данных и систем реального времени. Используя множество потоков, такие приложения могут выполнять несколько задач одновременно, повышая пропускную способность и уменьшая время отклика.

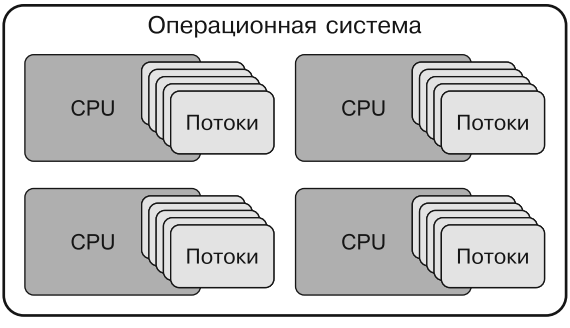

Рис. 1.1. Выполнение потоков разными CPU

Параллелизм и конкурентность

Термины *параллелизм (parallelism)* и *конкурентность (concurrency)* часто используются как синонимы, хотя они обозначают разные концепции. Параллелизм подразумевает одновременное выполнение нескольких задач, для чего требуется несколько единиц обработки (например, два или более ядер CPU). Конкурентность же связана с разработкой программ, отдельные части операций в которых могут перекрываться, даже если они не всегда выполняются одновременно. Параллелизм можно сравнить с работой нескольких строителей, бок о бок укладывающих кирпичи в стену дома, а конкурентность — с работой повара, который ловко управляется с приготовлением нескольких блюд. Теперь добавим еще поваров — суммарная производительность вырастет, но работа не обязательно станет параллельной: один повар может резать лук, а другой — ставить жаркое в духовку. Конечная цель остается неизменной — приготовить еду, возможно, из нескольких блюд.

Понятие потоков лежит в основе всей экосистемы Java и определяет принципы ее функционирования. Это фундамент, на котором построены многие мощные функции и инструменты. Даже функции, которые большинство программистов воспринимает как данность, без потоков были бы невозможны. Будь то система сборки мусора, решающая проблему управления памятью в Java, или выполнение простейшей Java-программы для вывода `Hello, World!` — в фоновом режиме работают потоки.

Приведу пример. Даже те, кто только знакомится с языком Java, используют потоки, сами того не зная. Казалось бы, какие потоки нужны для строки кода,

с которой начинается большинство учебников по программированию. Но JVM (Java Virtual Machine) выполняет данный код в потоке, который принято называть *главным потоком программы (main thread)*:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
    // Вывод имени потока, выполняющего метод main
    System.out.println("Executed by thread: "
      + Thread.currentThread().getName());
  }
}
```

Запустим эту программу и получим имя потока, выполняющего метод `main`, как правило — тоже `main`:

```
Hello, World!
Executed by thread: main
```

Данный пример показывает, что даже самые простые Java-программы работают в парадигме потоков. Это влечет за собой серьезные последствия: Java-разработчики — сознавая это или нет — используют мощь потоков практически на каждом этапе своей работы, начиная от запуска элементарных программ и заканчивая применением продвинутых механизмов вроде сборки мусора.

Таким образом, потоки — обязательный элемент Java: именно они делают его языком, способным масштабироваться для работы с крупными базами данных и распределенными системами.

Потоки — основа программной платформы Java

Потоки интегрированы во все уровни программной платформы Java и играют ключевую роль не только в простом выполнении кода. Например, они лежат в основе обработки исключений, отладки и профилирования Java-приложений.

Исключения и потоки

В Java каждый поток имеет собственный стек вызовов, который фиксирует все вызовы методов в течение жизненного цикла потока. При возникновении исключения этот стек становится важной частью диагностической информации: он отображает последовательность вызовов методов, которые привели к исключению, — это помогает разработчикам выявить первопричину проблемы. Рассмотрим тот же пример:

```
import java.sql.SQLException;

public class CallStackDemo {
  public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(CallStackDemo::processOrder);
```

```
    thread.setName("mcj-thread");
    thread.start(); thread.join();
  }

  static void processOrder() {
    validateOrderDetails();
  }

  static void validateOrderDetails() {
    checkInventory();
  }

  static void checkInventory() {
    updateDatabase();
  }

  static void updateDatabase() {
    try {
      throw new SQLException("Database connection error");
    } catch (SQLException e) {
      throw new InventoryUpdateException("Database Error: " +
        "Unable to update inventory", e);
    }
  }
}

class InventoryUpdateException extends RuntimeException {
  public InventoryUpdateException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

В данном сценарии операция обновления базы данных завершается с ошибкой, которая распространяется вверх по стеку вызовов. Основной поток начинает с вызова метода `processOrder`, вызывающего `validateOrderDetails`, который вызывает `checkInventory`, а тот, в свою очередь, — `updateDatabase`, который выбрасывает исключение. В консоли мы увидим такой вывод:

```
Exception in thread "mcj-thread" ca.bazlur.mcj.chap1.InventoryUpdateException:
Database Error: Unable to update inventory
  at ca.bazlur.mcj.chap1.CallStackDemo.updateDatabase(CallStackDemo.java:33)
  at ca.bazlur.mcj.chap1.CallStackDemo.checkInventory(CallStackDemo.java:26)
  at ca.bazlur.mcj.chap1.CallStackDemo.
  validateOrderDetails(CallStackDemo.java:22)
  at ca.bazlur.mcj.chap1.CallStackDemo.processOrder(CallStackDemo.java:18)
  at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: java.sql.SQLException: Database connection error
  at ca.bazlur.mcj.chap1.CallStackDemo.updateDatabase(CallStackDemo.java:31)
  ... 4 more
```

Этот стек-трейс позволяет разработчикам изучить стек вызовов в контексте конкретного потока (в данном случае потока `mcj-thread`, где возникло

исключение). Такая гранулярность крайне полезна в JVM, особенно при отладке многопоточных приложений, где несколько потоков могут выполняться конкурентно для решения одинаковых или различных задач. Эта информация делает отладку более целенаправленной и эффективной, помогает быстрее выявлять проблемы и в итоге ускоряет их устранение.

Отладчик и потоки

Каким образом отладчик Java определяет, где именно приостановить выполнение работающего приложения? Ответ снова кроется в потоках. Подключаясь к различным потокам приложения, отладчик может выбрать, какой из них проверить или даже какому изменить состояние во время отладки активного приложения. Это критически важно для поиска и исправления ошибок в приложениях, где несколько потоков способны одновременно выполняться на разных этапах, а их взаимодействие требует анализа.

В контексте потоков доступны операции перемещения по коду: шаг со входом (step into), шаг с обходом (stepped over) и шаг с выходом (step out). В одной сессии отладки могут инспектироваться один или несколько независимых стеков вызовов активных потоков. Стек вызовов показывает историю событий, например, какие методы привели поток к текущей точке программы. Когда пытаешься обнаружить причину неожиданных исключений или ошибочных манипуляций с данными, очень важно понять, как поток пришел к определенному состоянию.

Профилировщик и потоки

Потоки равно важны как для понимания механизма управления задачами в Java, так и для профилирования. Их применение способствует высочайшей точности при отладке и обеспечивает гранулярный подход при профилировании. Фактически потоки являются основой анализа производительности в Java. Инструменты профилирования используют информацию о потоках, чтобы показать детальную картину работы приложения: выявляют узкие места, помогают диагностировать сложные проблемы синхронизации в многопоточном коде и находить способы ускорения его работы. Одним словом, потоки — это ключ к пониманию и повышению производительности многопоточных Java-приложений.

Потоки в Java играют важную роль во многих операциях (например, в диагностике, отладке и профилировании), предоставляя детальную аналитическую информацию о выполнении программ на уровне отдельных потоков. Хотя разработчики не используют потоки напрямую, те работают в фоновом режиме, усиливая преимущества Java-приложений. Примеры таких потоков: потоки сборщика мусора для очистки памяти; потоки компиляции в JIT-системе для повышения производительности; диспетчер сигналов, финализаторы

и обработчики ссылок для бесперебойной работы JVM, а также потоки виртуальной машины и сервисные потоки для основных задач JVM и диагностики. Можно смело утверждать, что потоки в Java представляют собой один из важнейших уровней программной платформы, и Java-разработчик обязан в них разбираться.

Зарождение потоков в Java 1.0

Выпущенная в 1996 году Java 1.0 изначально поддерживала потоки. Эта ключевая особенность выделяла Java среди многих языков того времени. Поначалу потоки можно было создавать только двумя способами: через наследование класса `Thread` или реализацию интерфейса `Runnable`. Например:

```
public class ThreadCreationDemo {
  public static void main(String[] args) {
    // Расширение класса Thread
    Thread t1 = new ThreadByExtension("Worker-1"); ❶
    t1.start();
    // Реализация интерфейса Runnable (классическая)
    Thread t2 = new Thread(
      new RunnableImplementation(), "Worker-2"); ❷
    t2.start();
    // Анонимный внутренний класс (Java 1.1)
    Thread t3 = new Thread(new Runnable() { ❸
      @Override
      public void run() {
        System.out.println("Anonymous: " +
          Thread.currentThread().getName());
      }
    }, "Worker-3");
    t3.start();
    // Лямбда-выражение (Java 8)
    Thread t4 = new Thread(() -> ❹
      System.out.println("Lambda: " +
        Thread.currentThread().getName()),
        "Worker-4");
    t4.start();
  }
}

class ThreadByExtension extends Thread {
  public ThreadByExtension(String name) {
    super(name);
  }
  @Override
  public void run() {
    System.out.println("Extended Thread: " + getName());
  }
}
```

```
class RunnableImplementation implements Runnable {
  @Override
  public void run() {
    System.out.println("Runnable: " +
      Thread.currentThread().getName());
  }
}
```

Рассмотрим ключевые принципы проектирования с самого начала по порядку.

❶ Наследование от класса `Thread` предлагает прямой доступ к методам потока, но это единичное наследование.

❷ Реализация `Runnable` — предпочтительный подход: определение задач отделяется от управления потоком, и сохраняется гибкость наследования.

❸ Анонимные внутренние классы, добавленные в Java 1.1, снизили объем boilerplate-кода для разовых задач.

❹ Лямбда-выражения (Java 8) сделали реализацию интерфейса `Runnable` еще лаконичнее, поскольку этот интерфейс — функциональный.

Этот базовый API оставался неизменным почти три десятилетия, что показывает устойчивость дизайна. Хотя в Java добавили множество решений для улучшения конкурентности, от `java.util.concurrent` до виртуальных потоков, потоки как исходные строительные блоки остаются основой многопоточной модели Java.

Запуск потоков

Независимо от способа создания потока его запуск всегда начинается с вызова метода `start` у объекта `Thread`. Это необходимый шаг, поскольку вызов `start` не только запускает метод `run`, но и выполняет подготовительные операции, такие как выделение системных ресурсов. После этого метод `start` вызывает метод `run` в новом потоке выполнения.

Крайне важно не вызывать метод `run` напрямую, поскольку это приведет к выполнению `run` в уже вызванном потоке, а не в новом.

Фреймворк Executor позволяет создавать потоки в современных Java-приложениях гораздо эффективнее, чем вручную через конструкторы. Исполнители (executors) абстрагируют процесс управления потоками, поддерживая пул потоков — группу заранее созданных потоков, готовых к выполнению задач. Этот подход существенно снижает затраты на создание потоков.

Когда задача передается исполнителю, она помещается в очередь. Затем один из потоков пула забирает задачу из очереди и выполняет ее. Такая организация

упрощает конкурентное программирование и оптимизирует использование ресурсов благодаря неоднократному использованию потоков.

Простой пример применения исполнителя для управления пулом потоков:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
  public static void main(String[] args) {
    try (ExecutorService executor = Executors.newFixedThreadPool(5)) {
      for (int i = 0; i < 10; i++) {
        final int taskId = i;
        executor.submit(() -> {
          System.out.println("Executing task " + taskId
          + " in thread " + Thread.currentThread().getName());
        });
      }
    }
  }
}
```

Скрытые затраты на потоки

Многие современные веб-приложения работают по модели thread-per-request, где для каждого запроса предназначен отдельный поток, который управляет всем жизненным циклом запроса/ответа. В качестве примера рассмотрим жизненный цикл запроса в типичном веб-приложении, работающем под управлением контейнера сервлетов (Apache Tomcat, Jetty или любого Java EE веб-сервера). *Контейнер сервлетов (servlet container)* — это приложение, отвечающее за обработку запросов к веб-приложению. Получив запрос, контейнер назначает его для обработки одному из потоков своего пула. Поток активируется, как бы говоря: «Эй, мне пришел запрос от пользователя. Теперь это моя задача, я ею займусь». По сути, поток отвечает за обработку запроса/ответа в течение всего жизненного цикла (рис. 1.2).

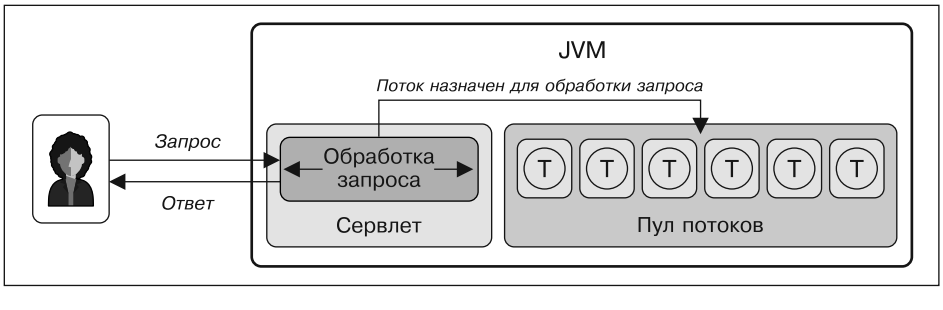

Рис. 1.2. Пул потоков для обработки запросов в сервлете

Эта модель особенно полезна при большом числе одновременных подключений, поскольку увеличение конкурентности, в свою очередь, повышает пропускную способность веб-приложения. Эта ключевая особенность модели thread-per-request позволяет современным веб-приложениям масштабироваться, чтобы эффективно справляться с растущим объемом запросов.

Большинство современных операционных систем способны обрабатывать миллионы конкурентных подключений, поэтому кажется разумным создавать еще больше потоков для увеличения пропускной способности. Из закона Литтла (Little's Law) можно сделать вывод: чем больше потоков в пуле, тем выше должна быть пропускная способность[1]. Однако на практике такая логика оказывается неоднозначной и не всегда приводит к ожидаемым результатам.

Пропускная способность (throughput) веб-приложений — это скорость обработки запросов и выдачи сервером ответов; обычно измеряется в запросах в секунду (RPS) или транзакциях в секунду (TPS). Она отражает способность приложения справляться с нагрузкой и служит количественным показателем для оценки производительности, масштабируемости и использования ресурсов. Пропускную способность можно рассчитать по формуле:

Пропускная способность = Общее количество обработанных запросов / Общее время.

Эта формула служит ориентиром при тестировании производительности, масштабируемости и прогнозировании нагрузки, а также помогает оптимально распределять ресурсы для обеспечения качественного пользовательского опыта.

Важно учитывать объем памяти, занимаемый каждым потоком: около 2 МиБ[2] (вне кучи на каждый поток). В крупномасштабных приложениях с тысячами конкурентных подключений можно быстро достичь значительного расхода памяти. Совокупный объем памяти, занятый потоками, существенно влияет на количество соединений, с которыми может работать приложение. Более того, если приложение исчерпает физическую память, начнется подкачка данных на диск — операция, которая выполняется значительно медленнее, чем обращение к RAM. Время чтения с диска может быть в 1000 раз больше, чем из RAM. Уже один этот фактор способен серьезно повлиять на производительность.

Кроме прочего, важно понимать, что потоки в Java представляют собой тонкую обертку над нативными потоками базовой операционной системы.

---

[1] Закон Литтла — это фундаментальный принцип теории очередей, применимый и к многопоточным приложениям. Он связывает три ключевых параметра производительности: задержку (latency), конкурентность (concurrency) и пропускную способность (throughput). В следующей главе мы детально рассмотрим этот закон, и вы увидите, как увеличение конкурентности повышает пропускную способность.

[2] Это стандартный размер для большинства Linux-окружений, однако в зависимости от операционной системы он может меняться.

Максимальное количество потоков, которое можно создать для приложения, фактически ограничено лимитом на создание нативных потоков, установленным базовой ОС. Почти все операционные системы имеют верхний предел числа создаваемых потоков — это узкое место для масштабирования приложения.

Помимо прочего, существуют затраты на переключение контекста. Создание потока — это нагрузка не только на память, но и на процессор. При переключении между потоками происходит смена контекста: необходимо сохранить состояние текущего потока и восстановить состояние нового. Все переключения контекста потребляют такты CPU, что при высокой нагрузке существенно влияет на производительность. Это еще один недооцененный источник нагрузки в вашем приложении.

Сколько потоков можно создать

Теперь рассмотрим, как измерить ограничение на создание потоков в вашей среде. Запустив простую тестовую программу, можно определить максимальное количество потоков, которое система способна обработать без возникновения проблем. Вот заготовка кода для начала:

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.LockSupport;

public class ThreadLimitTest {
  public static void main(String[] args) {
    var threadCount = new AtomicInteger(0);
    try {
      while (true) {
        var thread = new Thread(() -> {
          threadCount.incrementAndGet();
          LockSupport.park();
        });
        thread.start();
      }
    } catch (OutOfMemoryError error) {
      System.out.println("Reached thread limit: " + threadCount);
      error.printStackTrace();
    }
  }
}
```

Выполним эту программу и посмотрим, сколько потоков можно создать, не исчерпав память и не достигнув других системных ограничений:

```
Reached thread limit: 16363
java.lang.OutOfMemoryError: unable to create native thread: possibly out of
memory or process/resource limits reached
  at java.base/java.lang.Thread.start0(Native Method)
  at java.base/java.lang.Thread.start(Thread.java:1526)
  at ca.bazlur.chapter0.ThreadLimitTest.main(ThreadLimitTest.java:15)
```

На моей машине ошибка `OutOfMemoryError` (https://oreil.ly/vD03r) возникла после создания 16 363 потоков. Этот эксперимент показывает, что существует предел числа создаваемых потоков, ограниченный в основном операционной системой и оборудованием. Соответственно, на вашей машине этот лимит может отличаться.

Как видите, потоки обеспечивают значительные преимущества для масштабирования веб-приложений, но при этом они связаны с определенными затратами. Эти, иногда неочевидные, затраты необходимо тщательно учитывать для достижения оптимальной производительности.

Эффективность использования ресурсов в высоконагруженных приложениях

Современные приложения часто должны обрабатывать большие объемы данных и одновременно обслуживать высокий входящий трафик. Эти условия создают серьезные проблемы для финансовой устойчивости, особенно когда облачные решения стали для многих компаний средой по умолчанию. Даже незначительная неэффективность использования ресурсов может привести к стремительному росту затрат. Поскольку количество доступных потоков ограничено, важно использовать их рационально. Однако на практике потоки часто блокируются, что снижает их эффективность.

Рассмотрим следующий пример, где на основе различных факторов рассчитывается персональная кредитоспособность:

```
public Credit calculateCredit(Long personId) {
  var person = getPerson(personId);         // Вызов БД блокирует поток
  var assets = getAssets(person);           // Вызов API блокирует поток
  var liabilities = getLiabilities(person); // Вызов БД блокирует поток
  importantWork();                          // Ресурсоемкие вычисления
  return calculateCredits(assets, liabilities);
}
```

В приведенном фрагменте кода показана последовательность из пяти вызовов методов, выполняющихся один за другим. Допустим, каждый вызов занимает 200 мс. Тогда время выполнения метода `calculateCredit()` составит около 1 с — примерно в пять раз больше (1000 мс = 200 мс × 5).

Основная проблема здесь заключается не в простое потока, а в его блокировке во время операций ввода-вывода. Когда поток выполняет `getPerson(personId)`, он делает запрос к базе данных и вынужден ждать ответа по сети. В этот период поток нельзя переназначить для обработки других запросов или выполнения иной работы. Аналогичная блокировка происходит при вызове `getAssets()` и `getLiabilities()`. Хотя технически поток «занят» выполнением этих методов,

большую часть времени он заблокирован операциями ввода-вывода, а не занимается продуктивными вычислениями.

Такое поведение с блокировками весьма неэффективно: ценные ресурсы потоков простаивают в ожидании ответов внешних систем, хотя могли бы обрабатывать другие запросы. В высоконагруженных приложениях это существенно ограничивает масштабируемость, так как каждый заблокированный поток означает, что для обработки входящих запросов стало одним потоком меньше.

Чтобы достичь высокой эффективности потоков, в частности минимизировать время блокировки при операциях ввода-вывода, в Java был внесен ряд важных изменений. Среди них: поддержка асинхронных вызовов методов (позволяющих методам выполняться независимо во время операций ввода-вывода), пулы потоков (неоднократное использование фиксированного числа потоков для выполнения задач) и, не так давно, современные реактивные модели программирования (основанные на неблокирующем событийно-ориентированном взаимодействии).

Начнем с классических методов работы с потоками, таких как ручное создание и управление отдельными потоками, а затем постепенно перейдем к современным предложениям.

Стратегия параллельного выполнения

В приведенном ранее фрагменте кода мы видим, что вызовы методов `getAssets()`, `getLiabilities()` и `importantWork()` не зависят друг от друга, так что их можно выполнять параллельно. Такой подход сократит время блокировки нашего главного потока, хотя и не устранит ее полностью.

Наша стратегия параллельного выполнения действительно уменьшает время простоя главного потока. Поскольку он меньше простаивает, то после завершения выполнения метода `calculateCredit()` быстрее освобождается для обработки других задач.

Начнем с создания базовых структур данных, которые понадобятся для нашего кредитного калькулятора:

```
// Модели для расчета кредита
record Credit(double score) {}
record Person(Long id, String name) {}
record Asset(String type, double value) {}
record Liability(String type, double amount) {}
```

Реализуем наш подход к параллельному выполнению:

```
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.atomic.AtomicReference;
```

```
Credit calculateCreditWithUnboundedThreads(Long personId)
                                    throws InterruptedException {

  var person = getPerson(personId);

  var assetsRef = new AtomicReference<List<Asset>>(); ❶
  var t1 = new Thread(() -> {
    var assets = getAssets(person);
    assetsRef.set(assets); ❷
  });

  var liabilitiesRef = new AtomicReference<List<Liability>>();
  Thread t2 = new Thread(() -> {
    var liabilities = getLiabilities(person);
    liabilitiesRef.set(liabilities);
  });

  var t3 = new Thread(() -> importantWork()); ❸

  t1.start(); ❹
  t2.start();
  t3.start();

  t1.join(); ❺
  t2.join();

  var credit = calculateCredits(assetsRef.get(), liabilitiesRef.get()); ❻

  t3.join(); ❼

  return credit;
}
```

Рассмотрим, как работает данный код.

❶ Для безопасного обмена данными между потоками используем `AtomicReference` (https://oreil.ly/zmP1k).

❷ Потокобезопасное присваивание результатов, доступных из основного потока.

❸ Независимые задачи могут выполняться параллельно, не влияя на расчет кредита.

❹ Все три потока запускаются конкурентно, сокращая общее время выполнения.

❺ Ожидается завершение потоков для активов и пассивов перед последующим расчетом кредита.

❻ Расчет кредита выполняется с использованием результатов параллельных операций.

❼ Обеспечивает завершение независимой работы перед возвращением методом значений.