Параллелизм и транзакции (часть 2)

Денис С. Мигинский

Транзакционная память

Механизм управления параллелизмом (concurrency control): механизм, обеспечивающий корректность результата в условиях параллелизма.

Транзакционная память (transactional memory): механизм управления параллелизмом на основе транзакций.

Программная реализация (Software TM):
Одна из первых: Glasgow Haskell Compiler (начало 90-х)
Clojure – заложена в концепцию языка
На текущий момент имеется практически для всех языков в форме внешних библиотек

Аппаратная реализация (Hardware TM):
IBM BlueGene/Q processor (2011 г.), первый коммерческий сервер (IBM) – 2012 г.

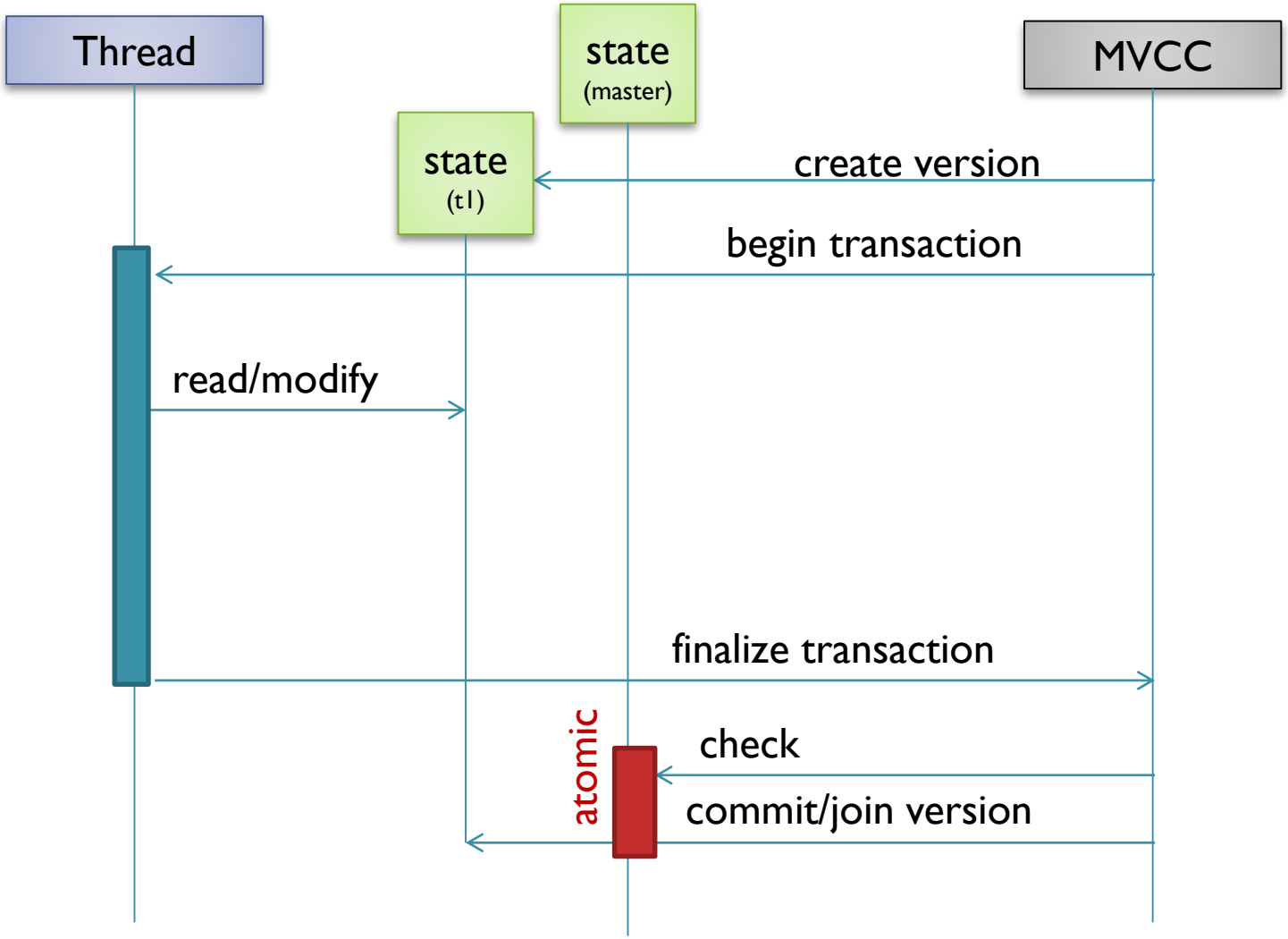
Multiversion Concurrency Control (MVCC)

Optimistic/Lock-free Concurrency Control (OCC): метод управления параллелизмом, предполагающий, что транзакции могут (хотя и не обязаны) завершиться корректно без блокировки. Конфликты проверяются непосредственно перед операцией **commit**.
(идея предложена H.T. Kung, 1981 г)

Противоположный подход: Pessimistic/Lock-based CC: метод априори предполагает «худшее», поэтому заранее блокирует все что может сломаться.

Версионный механизм управления параллелизмом (MVCC): частный случай транзакционной памяти, где изоляция транзакций осуществляется посредством создания различных представлений (view) данных, реализуя таким образом OCC

Принцип действия STM/MVCC (идеальный случай)



Управляемая (STM-managed) ссылка в Clojure

```
;конструирование ссылок
(def mobj1 (ref 1))
(def mobj2 (ref 2))

(let [swap-fn
      (fn []
;объявление транзакции
        (dosync
          (let [v1 @mobj1
                v2 @mobj2]
            (ref-set mobj1 v2)
            (ref-set mobj2 v1))))
```

…

Запуск транзакции

```
(let [swap-fn …
      t1 (new Thread swap-fn)
      t2 (new Thread swap-fn)]
  (.start t1)
  (.start t2)
  (.join t1)
  (.join t2))
(println @mobj1 @mobj2)

>> 1 2
```

Операции с ref

```
;конструирование
(ref val)

;аналогично reset! для atom
(ref-set ref new-val)

;аналогично swap! для atom
(alter ref f & args)

;аналогично alter, но с требованием
;коммутативности от f
(commute ref f & args)
```

Попытка чернового чтения

```
(def cnt1 (ref 1))
(def cnt2 (ref 1))

(dosync
  (Thread/sleep 80)
  (println "t1(before): " @cnt2)
  (alter cnt1 inc)


  (Thread/sleep 100)
  (println "t1(after): " @cnt2)

  (Thread/sleep 100))
```

```
(dosync
 (Thread/sleep 100)


 (println "t2(before): " @cnt1)
 (alter cnt2 inc)
 (Thread/sleep 100)

 (println "t2(after): " @cnt1)
 (Thread/sleep 100))
```

```
>>
t1(before): 1
t2(before): 1
t1(after): 1
t2(after): 1
```

Попытка невоспроизводимого чтения

```
(def cnt1 (ref 1))
(def cnt2 (ref 1))

(dosync
  (Thread/sleep 80)
  (println "t1(before): " @cnt2)
  (alter cnt1 inc)


  (Thread/sleep 100)
  (println "t1(after): " @cnt2)

  #_(Thread/sleep 100))
```

```
(dosync
 (Thread/sleep 100)


 (println "t2(before): " @cnt1)
 (alter cnt2 inc)
 (Thread/sleep 100)

 (println "t2(after): " @cnt1)
 #_(Thread/sleep 100))
```

```
>>
t1(before): 1
t2(before): 1
t1(after): 1
t2(before): 2
t2(after): 2
```

Обеспечение целостности ссылки

```
;аналогичный механизм доступен для atom
(let [acc (ref 50 :validator #(>= % 0))]
  (dosync (alter acc #(- % 30)))
  (println "sum:" @acc)
  (dosync (alter acc #(- % 30)))
  (println "sum:" @acc))
```

```
>>
sum: 20
IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
```

Коммутативные изменения: проблема

```
(def cnt1 (ref 1))
```

```
(dosync
  (println "t1 started")
  (alter cnt1 inc)
  (Thread/sleep 100))
```

```
(dosync
  (println "t2 started")
  (alter cnt1 inc)
  (Thread/sleep 100))
```

```
;возможный вывод
>>
t1 started
t2 started
t1 started
t1 started
t1 started
```

Решение: commute

```
(def cnt1 (ref 1))
```

```
(dosync
  (println "t1 started")
  (commute cnt1 inc)
  (Thread/sleep 100))
```

```
(dosync
  (println "t2 started")
  (commute cnt1 inc)
  (Thread/sleep 100))
```

```
;вывод (с точностью до
;перестановки)
>>
t1 started
t2 started
```

Вскрытие commute

```
(def cnt1 (ref 1))

(defn debug-inc [x]
  (println "debug: inc")
  (inc x))

(dosync
  (println "t1 started")
  (commute cnt1 debug-inc)
  (Thread/sleep 100)
  (println "t1 finished"))
```

```
(dosync
  (println "t2 started")
  (commute cnt1 debug-inc)
  (Thread/sleep 100)
  (println "t2 finished"))
```

```
;вывод (с точностью до перестановки)
>>
t1 started
debug: inc
t2 started
debug: inc
t1 finished
debug: inc
t2 finished
debug: inc
```

Проблемы с commute

```
(def id-seq (ref 0))
```

```
(dosync
   (let [id (commute id-seq inc)]
     (println "t1: uniq id is" id)
     (Thread/sleep 100)))
```

```
(dosync
   (let [id (commute id-seq inc)]
     (println "t2: uniq id is" id)
     (Thread/sleep 100)))
```

```
>>
t2: uniq id is 1
t1: uniq id is 1
```

Замена **commute** на **alter** решает проблему

Выводы:

- **commute** можно использовать для коммутативных и быстрых операций
- транзакция не должна опираться на результаты **commute**
- не уверен – используй **alter**

Ложка дегтя

Есть два банковских счета. Допускается, что один из них может уйти в «минус» при условии что сумма счетов положительна. Запускаем параллельно следующие транзакции:

```
(def acc1 (ref 50))
(def acc2 (ref 50))
(def sum 70)
```

```
(dosync
  (when (>= (+ @acc1 @acc2) sum)
    (Thread/sleep 100)
    (alter acc1 #(- % sum)))
  (Thread/sleep 100))
```

```
(dosync
  (when (>= (+ @acc1 @acc2) sum)
    (Thread/sleep 100)
    (alter acc2 #(- % sum)))
  (Thread/sleep 100))
```

```
;после завершения обеих транзакций
>>
acc1: -20
acc2: -20
```

Защита с помощью фиктивной записи

```
(dosync
  ;;фиктивная запись
  (alter acc1 identity)
  (alter acc2 identity)
  (when (>= (+ @acc1 @acc2) sum)
    (alter acc1 #(- % sum))))

(dosync
  ;;более правильное решение
  (ensure acc1)
  (ensure acc2)
  (when (>= (+ @acc1 @acc2) sum)
    (alter acc1 #(- % sum))))
```

Замечание: данная проблема специфична для реализации STM в Clojure. Однако «идеальных» реализаций не бывает (или они очень ограничивают параллелизм). Для каждой реализации транзакционного механизма очень желательно изучить его особенности перед серьезным использованием.

Преимущества и недостатки транзакций

Преимущества транзакций (lock-free):

- Потенциально более высокий уровень параллелизма
- Обеспечение SoC: не нужно явно заботиться о защите ресурсов
- **Низкий уровень сокрытия ошибок**
- Множество вариантов реализации/оптимизации под конкретные задачи

Недостатки:

- Дополнительные накладные расходы
- Полная изоляция транзакций может свести на нет более высокий уровень параллелизма
- Неполная изоляция транзакций требует знания особенностей того или иного механизма управления транзакциями (что также несколько нарушает SoC)
- Границы транзакций невозможно рассматривать как отдельный изолированный аспект

State of the art

Многопоточность, как инструмент для решения «элитарных» задач	Многопоточность в каждый дом!
Блокировки	Транзакции
Изменяемое состояние, как основной механизм абстракции	Изоляция и минимизация изменяемого состояния и побочных эффектов
Структурное/императивное программирование	Функциональное программирование

В таком виде транзакции хоть и дают существенные преимущества, но всех проблем не решают.

Куда двигаться дальше?

Copy-modify-merge

• Понятие транзакции сохраняется

• Основные положения MVCC сохраняются: транзакция взаимодействует с изолированной копией (версией) состояния

• По завершении транзакции, если публичная версия состояния была изменена, транзакция не повторяется, вместо этого состояния объединяются по заданной стратегии

• Откат транзакции происходит только в случае возникновения неразрешимых конфликтов (нарушение целостности, либо расхождение версий не может быть разрешено стратегией объединения)

• Требование коротких транзакций отменяется, либо становится менее значительным

Примеры использования: CVS, SVN и т.д.

Идея «версионной» памяти (concurrent revisions)

• Управление параллелизмом с помощью концепции copy-modify-merge

• Откат транзакции из-за возникновения race-condition не должен происходить никогда (тогда можно снять требование отсутствия дополнительных побочных эффектов). *В более мягкой постановке повтор все же происходить может – гибрид с MVCC.*

• Пользователь должен иметь возможность задавать различные стратегии объединения

• Допускаются иерархические транзакции (версии состояния образуют дерево)

Транзакции в распределенных системах

- Распределенная система =>
  распределенное состояние =>
  обеспечение «распределенной» целостности =>
  распределенная транзакция

- Примеры: SVN, банковские транзакции

- Дополнительные проблемы: невозможность выполнить транзакцию в одном потоке, более высокая вероятность сбоев (узлов, сети и т.д.)

Иллюстрация: Two-Phase Commit Protocol (2PC)

Протокол должен учитывать, что в любой момент может произойти сбой (порваться связь и т.д.)

Распределенные транзакции: state of the art

- Не существует протокола, гарантирующего в 100% случаев ACID. В некоторых (хоть и редких) случаях требуется вмешательство человека
- Накладные расходы по обеспечению распределенных транзакций очень высоки.
- Способы борьбы с проблемами: изоляция состояния; stateless-сервисы, как парадигма stateful – как исключение. Проще говоря: всеми силами избегать распределенных транзакций.

CAP-теорема

В распределенной системе из трех характеристик

- Consistency (целостность)
- Availability (доступность) – возможность интерактивного отклика
- Partition Tolerance (устойчивость к расщеплению) – работоспособность в условиях потери связности системы

возможно обеспечение только двух.

Интерпретация CAP

- CA – система, обеспечивающая надежность и доступность данных, но нетерпимая к расщеплению. В этом случае надежность должна быть обеспечена внешними средствами (как правило, intranet-системы, кластерные решения и т.д.)

- CP – тяжеловесные распределенные ACID-системы (например, банковские)

- AP – а так уж она нужна, эта целостность? Давайте разрешим иногда версиям состояния на разных узлах расходиться, транзакциям иногда драться!
  - пользователь разок не попадет на свою страницу (DNS);
  - исходные коды иногда можно и руками объединить (SVN);
  - ну купят два клиента одну и ту же последнюю книгу – подарим одному бонусную карту (Amazon.com и т.д.).

Optimistic replication

Стратегия репликации, при которой реплики могут временно расходиться. Конфликты реплик также могут разрешаться после (а не в момент) транзакции

Противоположный подход (pessimistic replication): система гарантирует синхронизацию всех реплик. При этом транзакции становятся распределенными, и тем сложнее, чем больше узлов в систем

Подробнее, например, здесь:
Ya. Saito, M. Shapiro Optimistic Replication //ACM Computing Surveys, Vol. V, No. N, 3 2005, Pages 1–44

BASE

Acid – кислота, Base – основание (химия)

- Basically Available – фокусируемся на доступности системы
- Soft State – состояние на любом узле может быть потеряно и вычислено заново
- Eventual Consistency - Если дать системе достаточно времени, то изменения в ней распространятся. А пока не распространяться можно потерпеть некоторое расхождение в репликах.

Применение: noSQL базы данных

Другие механизмы управления параллелизмом

- Агент (agent) – активный объект (т.е. реально или виртуально владеющий потоком), способный принимать сообщения и реагировать на них в своем контексте

- Future(promise) – форма отложенных вычислений, объект, содержащий обещание вычислить значение. В некоторый момент обещание обращается в реальное значение, вычисления происходят в отдельном потоке (в отличие от **delay**).
  Удобно для «мелкозернистого» параллелизма.

Асинхронный примитив: agent

```
(let [mobj (agent 1)]
  (send mobj inc)
  (println @mobj)
  (Thread/sleep 100)
  (println @mobj))

>> 1
   2
```

Операции с agent

```
;конструирования
(agent val & opt)

;посылка сообщения, аналог swap!,
;но исполнится в отдельном потоке
(send agent f & args)

;ожидание обработки сообщений
;из данного потока
(await & agents)
(await-for timeout-ms & agents)
```

future

```
;;;В Java аналог требует явного запуска
;;;через ThreadPoolExecutor
(let [foo (future
            (Thread/sleep 100)
            0)]
  ;;join
  @foo)

>> 0
```

promise

```
;;;promise – гибрид let и future
;;;обещание, что кто-то «положит» туда
;;;значение (единственный раз!)

(let [foo (promise)]
  (deliver foo 1)
  @foo)

>> 1
```