Содержание

1 Введение

При разработке современного веб-сервиса необходимо уделять особое внимание его быстродействию и надежности, особенно когда речь идет о веб-сервисаз, помогающих в ведении бизнеса (взаимодействие с банками, электронная отчетность и т.д.). Отказ подобных систем может повлечь финансовые потери клиента, а порой и поставить под угрозу весь его бизнес. Например, невовремя или неверно сданная отчетность может повлечь серьезные штрафы со стороны государства, вплоть до отзыва лицензии.

Отказ системы может произойти по разным причинам, например из-за сетевых неполадок или аварийного завершения работы одной из компонент. Например, если сервис, отвечающий за поиск по пользовательским данным, перестанет отвечать, то многие клиенты потеряют возможность полноценной работы в системе, поскольку лишаться возможности быстро находить нужную информацию. Эту проблему можно решить с помощью запуска сервиса на нескольких машинах: в таком случае если одна из реплик сервиса выйдет из строя, остальные смогут продолжить обработку запросов. Также данное решение может избавить нас от проблем с недоступностью из-за сетевых неполадок: выпадение одной реплики сервиса из сети не остановит работу системы.

По тем же причинам необходимо использовать отказоустойчивую базу данных. В зависимости от типов запросов, которые база данных должна уметь обрабатывать, и нагрузок, которые она должна выдерживать, можно выбрать одно из множества доступных решений: MongoDB, Berkeley DB (а также любое другое NoSQL решение) или же взять обычный MySQL с master-slave репликацией.

При разработке многопоточных приложений часто встает задача предоставления исключительного доступа к какому-либо разделяемому ресурсу. Рассмотрим простой пример: снятие денег со счета с двух банкоматов. Предположим, что на счете лежит 200 условных единиц, и два пользователя хотят снять с этого счета по 150 единиц в один и тот же момент времени. Алгоритм снятия денег в первом приближении достаточно прост: узнать

сколько денег на счете, и если средств достаточно, то выдать требуемую сумму и списать ее со счета. Если этот алгоритм будет выполняться одновременно двумя потоками без каких-либо блокировок, то может возникнуть следующая ситуация: оба потока одновременно запросят остаток на счете, поймут что средств достаточно и каждый снимет со счета по 150 условных единиц, оставив на счете отрицательное значение, что в действительности неприемлимо. Если же перед началом выполнения мы можем сказать, что сейчас мы будем работать со счетом и никто другой этого делать не должен, то такой ситуации получиться не может.

Большинство современных языков программирования содержит механизмы, позволяющие решить эту задачу, но только в рамках одного процесса, в некоторых языках реализованы механизмы для решения этой задача и для разных процессоров, запущенных на одном сервере. Однако когда речь идет о процессах, запущенных на разных серверах, возникает задача распределенной блокировки — необходим механизм, позволяющий решать проблему исключительного доступа к ресурсу с разных серверов.

Существует множество готовых решений данной задачи. Существует Chubby от Google, но оно является проприетарным и его исходного кода нет в открытом доступе. Есть Apache ZooKeeper, это решение лежит в свободном доступе и активно развивается, но оно обладает достаточно высоким порогом входа: более-менее удобный и надежный клиент для ZooKeeper есть только для Java, а реализация своей обертки — очень трудоемкая задача. Можно воспользоваться ресурсами любой централизованной базы данных с транзакциями (MySQL, MSSQL), но это решение будет иметь недопустимо низкую производительность.

В магистерской работе 2012 года Федора Фоминых задача о распределенной блокировке всплывает как подзадача при построении распределенной очереди на Cassandra (NoSQL база данных, которая активно используется у нас в проекте). Однако в ходе использования предложенного алгоритма была отмечена достаточно важная проблема — проседание производительности в случае попытки взятия одной и той же блокировки

достаточно большим количеством потоков.

Была поставлена цель — предложить и реализовать алгоритм, не имеющий проблемы со снижением производительности при любом количестве потоков. Цель была достигнута, на данный момент решение находится на стадии внедрения в проект.

2 Кратко о Cassandra

Apache Cassandra — распределенная система управления базами данных, относящаяся к классу NoSQL. За счет отказа от реляционности и транзакционности в NoSQL системах достигаются возможности хорошей горизонтальной масштабируемости и репликации. Это направление в компьютерных науках сейчас находится в стадии активного развития, и практически у каждой крупной IT-компании есть своя NoSQL база данных. Cassandra изначально была разработкой Facebook, однако в 2009 году было решено отдать проект фонду Apache Software.

В первом приближении на Cassandra можно смотреть как на следующие сущности (в порядке вложенности):

1. Кластер — множество серверов, на которых хранится множество баз данных;
2. Пространство ключей — база данных, множество таблиц;
3. Семейство колонок — таблица, множество элементов;
4. Колонка — ячейка, хранящая в себе конкретную запись.

Колонка содержит в себе следующую информацию:

1. Имя строки, в которой лежит ячейка;
2. Имя колонки, в которой лежит ячейка;
3. Набор байтов с хранимой информацией;
4. Время создания ячейки;
5. Время жизни ячейки.

Фактически семейство колонок — разреженная таблица, в которой каждая строка содержит множество ячеек, упорядоченное по имени колонки в

лексикографическом порядке. Порядок колонок в строке — важная особенность хранения данных, она является ключевой для предлагаемого алгоритма.

3 Требования к интерфейсу сервиса распределенной блокировки

Сформулируем требования к интерфейсу сервиса распределенной блокировки. Фактически необходимо реализовать два стратегии блокировки:

- Обязательная блокировка — подразумевает, что блокировка будет взята в любом случае: поток в любом случае дождется освобождения нужного ресурса и обязательно возьмет блокировку;
- Мягкая блокировка — подразумевает, что блокировка может быть и не взята: поток попытается взять блокировку, но если ресурс уже занят другим потоком, он продолжит свое выполнение.

Разница достаточно проста и прозрачна — первая стратегия применяется в случае когда оба потока обязательно должны совершить свое действие (например, вывести сообщение на консоль или в лог), вторая же применима если нам достаточно чтобы хотя бы один поток совершил свое действие (например, взял конкретную запись из некоторого хранилища и обработал ее).

Опишем данные требования на языке C#:

Листинг 1: Описание интерфейса

```
public interface IRemoteLockCreator
{
  IRemoteLock GetLock(string lockId); // обязательная блокировка
  bool TryGetLock(string lockId, out IRemoteLock remoteLock); // мягкая блокировка
}

public interface IRemoteLock : IDisposable
{
  public string ThreadId { get; }
  public string LockId { get; }
}
```

Здесь lockId — строковый идентификатор ресурса, доступ к которому необходимо получить. Метод Dispose в реализацях интерфейса IRemoteLock должен освобождать блокировку, в таком случае пользоваться сервисом блокировок будет достаточно удобно с помощью конструкции using:

Листинг 2: Использование конструкции using

```
...
using(remoteLockCreator.GetLock(lockId))
{
  // действия в блокировке
}
```

4 Обзор существующего решения

Рассмотрим алгоритм, описанный в [ссылка].

Заведем в Cassandra 2 таблицы: основную и теневую. В качестве основы для реализаций стратегий блокировок предлагается алгоритм, который пытается взять блокировку и возвращает в качестве результата одно из трех состояний:

- Success — поток успешно взял блокировку;
- AnotherThreadIsOwner — другой поток уже владеет блокировкой;
- ConcurrentAttempt — поток не смог взять блокировку, так как другой поток попытался сделать это одновременно с ним.

Сам алгоритм выглядит следующим образом:

Листинг 3: Алгоритм Cassandra.TryLock(lockId, threadId)

```
1. Взять ячейки из основной таблицы из строки lockId
2. Если ячейка одна:
3.   Если columnKey = threadId:
4.     Вернуть Success
5.   Иначе:
6.     Вернуть AnotherThreadIsOwner
7. Добавить ячейку в теневую таблицу
8. Взять ячейки из теневой таблицы
9. Если ячейка в теневой таблице одна
10.   Если нет ячеек в основной таблице
11.     Добавить ячейку в основную таблицу
12.     Удалить ячейку из теневой таблицы
13.     Вернуть Success
14. Удалить ячейку из теневой таблицы
15. Вернуть ConcurrentAttempt
```

С использованием этого алгоритма достаточно просто реализовать, например, стратегию обязательной блокировки:

Листинг 4: Алгоритм Cassandra.GetLock(lockId, threadId)

```
1. Присвоить attempt = 1
2. Вызвать Cassanrda.TryLock(lockId, threadId)
3. Если Success:
4.   Закончить
5. Если AnotherThreadIsOwner
6.   Подождать случайный промежуток времени от 0 до 1000 мс
```

```
 7.    Перейти к 2
 8. Если ConcurrentAttempt:
 9.    Подождать случайный промежуток времени от 0 до 50*attempt мс
10.    Присвоить attempt = attempt + 1
11.    Перейти к 2
```

По факту потоки будут бороться друг с другом за право захвата блокировки, не пытаясь договориться друг с другом. Теоретически при большом количестве потоков может сложиться такая ситуация, что потоки могут долгое время пытаться оказаться одни в теневой таблице, то есть время взятия блокировки хотя бы одним потоком становится непредсказуемым. Еще один недостаток этого алгоритма менее очевидный. Представим себе ситуацию, когда два потока последовательно много раз пытаются взять одну и ту же блокировку. Как только один из них сможет ее захватить, второй поток не сможет взять блокировку после первого же ее освобождения: возможно в момент отпускания блокировки первым потоком второй поток будет находиться в состоянии ожидания, в таком случае первый поток тут же этим воспользуется и захватит блокировку повторно. Таким образом второй поток не сможет захватить блокировку пока не выйдет из состояния ожидания в нужный момент. Это приводит к тому, что потоки будут выполнять свои действия неравномерно — сначала большую пачку действий выполнит первый потом, потом второй, потом первый вернет себе лидерство и так далее. В идеале же хотелось бы несколько более справедливый алгоритм, который не будет неявным образом отдавать предпочтение тому или иному потоку на протяжении долгого времени.

5 Описание алгоритмов

Желание получить алгоритм, справедливо распределящий доступ к разделяемому ресурсу между потоками, подводит к достаточно простой идее использования некоторого подобия очереди потоков. Для каждого идентификатора разделяемого ресурса будем хранить в Cassandra две строки: первую будем использовать в качестве подобия очереди (далее «очередь»), во второй будет храниться идентификатор потока, владеющего ресурсом (далее «основная строка»). При добавлении потока в очередь мы будем добавлять новую ячейку в строчку с очередью, при этом имя колонки этой ячейки будет подсчитано следующим образом:

Листинг 5: Определение имени колонки для ячейки в очереди

```
1. Положить в переменную currentTime строковое значение текущего времени в микросекундах
2. Добавлять к значению currentTime ведущие нули до тех пор, пока его длина не станет равной 20
3. Назначить именем колонки конкатенацию значений currentTime и threadId
```

В силу свойства порядка хранения ячеек в строке раньше всех в очереди окажется тот поток, который будет добавлен в очередь раньше, в случае равенства времен ближе к вершине очереди окажется поток с лексикографически меньшим идентификатором. При добавлении потока в основную строку именем колонки будем считать просто идентификатор потока.

Все доказательства основываются на том предположении, что потоки будут поступать в очередь с неубывающим временем в качестве префикса имени колонки. Последее предположение является достаточно грубым в условиях распределенной системы с множеством серверов, так как время на машинах может рассинхронизироваться в случае недобросовестного администрирования системы. Эта проблема была решена с помощью реализации сервиса времени.

5.1 Алгоритм обязательной блокировки

Листинг 6: Алгоритм Cassandra.GetLock(lockId, threadId)

```
1.  Добавить поток в очередь
2.  Если наш поток - первый в очереди:
```

```
 3.     Добавить ячейку в основную строку
 4.     Если в основной строке есть только одна запись:
 5.       Закончить
 6.     Иначе:
 7.       Удалить ячейку из основной строки
 8.       Перейти к шагу 2
 9.   Иначе:
10.     Перейти к шагу 2
```

Теорема 1. *В ходе выполнения алгоритма Cassandra.GetLock несколькими потоками блокировка не будет взята более чем одним потоком одновременно.*

Теорема 2. *Если в одно и то же время в очередь может поступить не более N потоков, то при одновременном выполнении алгоритма Cassandra.GetLock несколькими потоками рано или поздно каждый поток сможет захватить блокировку, обратившись к основной строке не более 2*N-1 раз.*

5.2 Алгоритм мягкой блокировки

Листинг 7: Алгоритм Cassandra.TryGetLock(lockId, threadId)

```
 1.   Если в основной строке есть хотя бы одна запись
 2.     Вернуть false
 3.   Добавить поток в очередь
 4.   Если наш поток - первый в очереди:
 5.     Добавить ячейку в основную строку
 6.     Если в основной строке есть только одна запись:
 7.       Вернуть true
 8.     Иначе:
 9.         Удалить ячейку из основной строки
10.       Перейти к шагу 4
11.   Иначе:
12.     Удалить поток из очереди
13.     Вернуть false
```

Утверждение 5.1. *В ходе выполнения алгоритма Cassandra.TryGetLock несколькими потоками блокировку сможет взять не более одного потока.*

Утверждение 5.2. *В ходе выполнения алгоритма Cassandra.TryGetLock несколькими потоками блокировку хотя бы один поток захватит блокировку.*

Теорема 3. *В ходе выполнения алгоритма Cassandra.TryGetLock несколькими потоками блокировку возъмет ровно один поток.*

5.3 Аренда и освобождение блокировки

Алгоритмы Cassandra.GetLock и Cassandra.TryGetLock позволяют захватить блокировку, при этом в результате захвата блокировки в очереди и в основной строке остаются колонки, порожденные потоком. Для освобождения блокировки достаточно удалить эти колонки из строчек. Эту логику достаточно поместить в метод Dispose у реализации интерфейса IRemoteLock, в таком случае при использовании конструкции using в конце исполнения кода внутри блокировки разделяемый ресурс освободится.

Однако давайте представим себе следующую ситуацию: поток успешно взял блокировку, после чего аварийно завершил свою работу таким образом, что метод Dispose не отработал. В итоге система находится в таком состоянии, что ни один поток не сможет взять блокировку на этот ресурс, и освободить его тоже никто не сможет, то есть блокировка захвачена навечно.

Можно решить эту проблему достаточно просто с использованием Cassanrda: при записи ячеек в очередь и в основную таблицу будем выставлять для этих ячеек TTL в размере N секунд, а в случае успешного взятия блокировки создадим дополнительный фоновый поток, который раз в $N/2$ секунд будет перезаписывать ячейки в строчках. Таким образом в случае аварийного завершения процесса, поток которого захватил блокировку и не успел ее освободить, ячейки исчезнут из таблицы, так как вместе с потоком, удерживающим блокировку, прекратит свою работу и поток, отвечающий за продление аренды.

Таким образом при успешном взятии блокировки нам необходимо создать фоновый поток, который будет время от времени перезаписывать существующие ячейки в строчках, а при освобождении блокировки нужно будет остановить этот фоновый поток и удалить записаные ячейки из очереди и из основной строки.

5.4 Сервис времени

Ранее упоминалось, что доказательства имеют право на существование в том случае, если время, с которым потоки поступают в очередь, не убывает. На практике обеспечить такое поведение в распределенной системе с использованием встроенных в язык средств практически невозможно: время на серверах может рассинхронизироваться, в результате чего несколько реплик могут убежать вперед на несколько минут, несколько реплик могут отстать на несколько минут. Для решения этой проблемы был реализован сервис, который гарантированно будет отдавать неубывающее значение времени.

Суть сервиса очень простая: он имеет несколько реплик, каждая из которых периодически пытается обновить значение текущего времени в специально заведенной под эти цели ячейке в Cassandra. При обновлении реплика берет локальное текущее время в микросекундах, сравнивает его со значением, лежащим в Cassandra, и если значение в ячейке меньше локального времени, то ячейка просто переписывается с новым значением.

5.5 Реализация решения

Описаные алгоритмы были реализованы на языке C#. Исходный код лежит в репозитории CassandraPrimitives на https://git.skbkontur.ru. Код покрыт тестами, проверяющими корректность поведения алгоритма в различных ситуациях. Также был реализован инструмент для измерения эффективности алгоритмов блокировок, в частности было проведено сравнение нового алгоритма со старым.

[Тут результаты бенчмарков и легенда к ним]

6 Планы на будущее

В ходе проектирования и реализации алгоритма были отмечен ряд проблем.

Корректность алгоритма в некоторой степени завязана на неубывании времени в системе, для чего пришлось реализовывать собственный сервис, который гарантирует неубывание. Но существует сценарий развития событий, при котором этот сервис будет отдавать одно и то же значение на протяжении нескольких минут: время на репликах рассинхронизировалось и на одной из них убежало вперед на 10 минут относительно остальных, после чего эта реплика вышла из строя. В результате на протяжении следующих 10 минут значение в ячейке обновляться не будет, так как оно больше чем локальное время на всех остальных репликах.

Проблема рассинхронизации времени также всплывает в логике аренды блокировки: в силу внутреннего устройства Cassandra если время на репликах разъедется больше чем на TTL, то ячейка будет тут же удаляться из строки, тем самым нарушив корректность работы алгоритма блокировки. Поэтому TTL приходится указывать достаточно большой (около 15 минут), что вообще говоря не очень хорошо по очевидным причинам.

В стадии разработки сейчас находится решение, в котором логика аренды блокировки лежит не на TTL, а на собственном сервисе, который следит за тем, что аренда продлевается. Однако в ходе реализации этого сервиса стали возникать проблемы, в ходе решения которых становится понятно, что в принципе логику с проверкой активности захватившего блокировку потока лучше решать на стороне хранилища. Таким образом в планах — реализация своего распределенного хранилища, заточенного исключительно под распределенные блокировки.