DINS®

Реализуем алгоритм “Token Bucket” на Java

О докладчике:

- Бухтояров Владимир. Опыт коммерческой разработки на Java 12 лет. Из них 7 лет в компании DINS.
- В настоящее время работаю в должности Java Lead Developer.
- Люблю заниматься параллельными алгоритмами и Grid Computing.
- Активно участвую в OpenSource.

Связь докладчика и темы:

Автор библиотеки Bucket4j - самой популярной реализации алгоритма Token Bucket для Java:

- 100 тысяч скачиваний с Maven Central ежемесячно.
- В зависимостях у 2200 проектов по данным github.
- Используется в JHipster, Armeria, Jobby, Kubernatess Java Client.

План доклада:

- Общий обзор задач ограничение пропускной способности.
- Примеры типичных проблем при реализации троттлинга.
- Обзор алгоритма Token Bucket.
- Реализуем Lock-Free Token Bucket на Java.
- Реализуем распределенную реализацию Token Bucket с использованием Apache Ignite.
- Весь код доступен на Github.

Типовые случаи, когда мы сталкиваемся с необходимостью ограничивать пропускную способность:

- Защита нашей системы от нагрузки, порождаемой запросами приходящими извне.
- Защита сторонних систем от нагрузки, которую порождает наша система.
- Реализация контрактных обязательств.
- Fraud Detection.

Защищаем API социальной сети

```
@Controller
public class AdvertisingMessagesEndpoint {

    private MessageService messageService;

    @RequestMapping(method = RequestMethod.POST, value = "/advertising/message")
    public void sendAdvertisingMessages(MessageBatch messages) throws RateLimitExceedException {
        int requiredPermits = messages.estimateCost();
        RateLimiter rateLimiter = SecurityService.getRateLimiter();
        if (!rateLimiter.tryAcquire(requiredPermits)) {
            throw new RateLimitExceedException();
        }
        messageService.send(messages);
    }

}
```

Минимально необходимый интерфейс:

```
public interface RateLimiter {

    boolean tryAcquire(int permits);

}
```

Реализуем самодельный RateLimiter:

- Пример кода.
- Разбор проблем.

Простой RateLimiter структуры данных

```
public class SimpleRateLimiter implements RateLimiter {

    public SimpleRateLimiter(long permits, Duration period) {
        this.availablePermits = permits;
        this.periodNanos = period.toNanos();
    }

    private final long periodNanos;
    private long availablePermits;
    private final LinkedList<IssuedPermits> issuedPermits = new LinkedList<>();

    private static final class IssuedPermits {
        private final long permits;
        private final long expirationNanotime;

        private IssuedPermits(long permits, long expirationNanotime) {
            this.permits = permits;
            this.expirationNanotime = expirationNanotime;
        }
    }
```

Простой RateLimiter алгоритм:

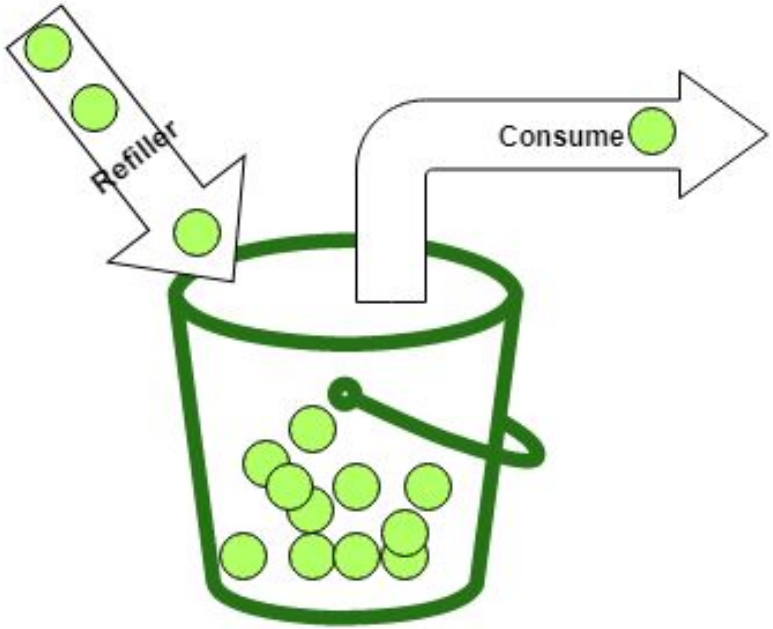
```
@Override
synchronized public boolean tryAcquire(int permits) {
    long nanoTime = System.nanoTime();
    clearPreviouslyIssuedPermits(nanoTime);

    if (availablePermits < permits) {
        // has no requested permits
        return false;
    } else {
        long expirationNanoTime = nanoTime + periodNanos;
        issuedPermits.addLast(new IssuedPermits(permits, expirationNanoTime));
        availablePermits -= permits;
        return true;
    }
}

private void clearPreviouslyIssuedPermits(long currentNanotime) {
    while (!issuedPermits.isEmpty()) {
        IssuedPermits issue = issuedPermits.getFirst();
        if (currentNanotime > issue.expirationNanotime) {
            availablePermits += issue.permits;
            issuedPermits.removeFirst();
        } else {
            return;
        }
    }
}
```

Простой RateLimiter проблема потребления памяти:

```
public SimpleRateLimiter(long permits, Duration period) {
    this.availablePermits = permits;
    this.periodNanos = period.toNanos();
}

private final long periodNanos;
private long availablePermits;
private final LinkedList<IssuedPermits> issuedPermits = new LinkedList<>();
```

Ищем алгоритм с фиксированной памятью:

Token Bucket: алгоритм с фиксированной памятью



Потребляемая память:

Объем ведра	8 байт
Текущее число токенов в ведре	8 байт
Число наносекунд на генерацию одного токена	8 байт
Заголовок объекта	16 байт
Итого:	40 байт

https://golb.hplar.ch/2019/08/rate-limit-bucket4j.html

DINS

Token Bucket: алгоритм с фиксированной памятью

The token bucket algorithm can be conceptually understood as follows:

- A token is added to the bucket every `1/r` seconds.
- The bucket can hold at the most M If a token arrives when the bucket is full, it is discarded.
- When a packet (network layer PDU) of *n* bytes arrives:
  - if at least *n* tokens are in the bucket, *n* tokens are removed from the bucket, and the packet is sent to the network.
  - if fewer than *n* tokens are available, no tokens are removed from the bucket, and the packet is considered to be *non-conformant*.

https://en.wikipedia.org/wiki/Token_bucket

Token Bucket сложность формулирования задачи:

Нам нужно:

- Число запросов - permits.
- За период времени - period.

Вместо этого нам дают:

- Capacity ведра - capacity.
- Скорость восполнения токенов - nanosToRefillOneToken.

Желаемое не совпадает с действительным, как с этим бороться???

Token Bucket сложность формулирования задачи:

Решение в лоб?

- *capacity = permits*
- *nanosToRefillOneToken = period/permits*

Не работает из-за наличия проблемы всплесков(burst).

Token Bucket проблема burst:

Вводная: имеем capacity=100 refill=100/1s(1/10ms).

1. В момент X имеем 100 токенов в ведре
2. В момент X забираем все 100 токенов, ведро пустое.
3. В момент X+1s ведро опять полно и содержит 100 токенов.
4. В момент X+1s опять забираем 100 токенов.

Итог: за одну секунду забрали 200 токенов.

Token Bucket сложность формулирования задачи:

Реальность:

Расплатой за фиксирование потребление памяти - является невозможность точно выразить исходный контракт *period/permits*

Что мы делаем когда требования реализовать нельзя:

- Не используем Token-Bucket??
- Переформулируем требования!
- **Осознанно** игнорируем несоответствие!

Реализуем Token Bucket на Java:

- Первоначальная реализация.
- Разбор проблем.

TokenBucket структуры данных

```
public class StraightforwardTokenBucket implements RateLimiter {

    public StraightforwardTokenBucket(long permits, Duration period,
              ScheduledExecutorService scheduler) {
        this.capacity = permits;
        this.availableTokens = capacity;
        long nanosToGenerateToken = period.toNanos() / permits;
        scheduler.scheduleAtFixedRate(this::addToken,
                nanosToGenerateToken, nanosToGenerateToken, TimeUnit.NANOSECONDS
        );
    }

    private final long capacity;
    private long availableTokens;
```

TokenBucket алгоритм:

```
private final long capacity;
private long availableTokens;

synchronized private void addToken() {
    availableTokens = Math.min(capacity, availableTokens + 1);
}

@Override
synchronized public boolean tryAcquire(int permits) {
    if (availableTokens < permits) {
        return false;
    } else {
        availableTokens -= permits;
        return true;
    }
}
```

TokenBucket проблема потребления CPU:

```
public StraightforwardTokenBucket(long permits, Duration period,
          ScheduledExecutorService scheduler) {
    this.capacity = permits;
    this.availableTokens = capacity;
    long nanosToGenerateToken = period.toNanos() / permits;
    scheduler.scheduleAtFixedRate(this::addToken,
            nanosToGenerateToken, nanosToGenerateToken, TimeUnit.NANOSECONDS
    );
}
```

SchedulerFreeTokenBucket структуры данных

```
public class SchedulerFreeTokenBucket {

    private final long capacity;
    private long availableTokens;

    private final long nanosToGenerateToken;
    private long lastRefillNanotime;

    public SchedulerFreeTokenBucket(long permits, Duration period) {
        this.nanosToGenerateToken = period.toNanos() / permits;
        this.lastRefillNanotime = System.nanoTime();

        this.capacity = permits;
        this.availableTokens = permits;
    }
```

SchedulerFreeTokenBucket алгоритм

```java
@Override
synchronized public boolean tryAcquire(int permits) {
    refill();
    if (availableTokens < permits) {
        return false;
    } else {
        availableTokens -= permits;
        return true;
    }
}

private void refill() {
    long now = System.nanoTime();
    long nanosSinceLastRefill = now - this.lastRefillNanotime;
    if (nanosSinceLastRefill <= nanosToGenerateToken) {
        return;
    }
    long tokensSinceLastRefill = nanosSinceLastRefill / nanosToGenerateToken;
    availableTokens = Math.min(capacity, availableTokens + tokensSinceLastRefill);
    lastRefillNanotime += tokensSinceLastRefill * nanosToGenerateToken;
}
```

SchedulerFreeTokenBucket проблема блокировок

```
@Override
synchronized public boolean tryAcquire(int permits) {
    refill();
    if (availableTokens < permits) {
        return false;
    } else {
        availableTokens -= permits;
        return true;
    }
}
```

LockFreeTokenBucket структура данных:

```
public class LockFreeTokenBucket {

    private final long capacity;
    private final long nanosToGenerateToken;
    private final AtomicReference<State> stateReference;

    private static final class State {

        private long availableTokens;
        private long lastRefillNanotime;

    }

    public LockFreeTokenBucket(long permits, Duration period) {
        this.nanosToGenerateToken = period.toNanos() / permits;
        this.capacity = permits;

        State initialState = new State();
        initialState.lastRefillNanotime = System.nanoTime();
        initialState.availableTokens = permits;

        this.stateReference = new AtomicReference<>(initialState);
    }
```

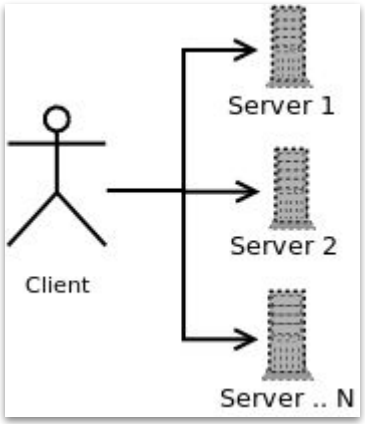
LockFreeTokenBucket алгоритм:

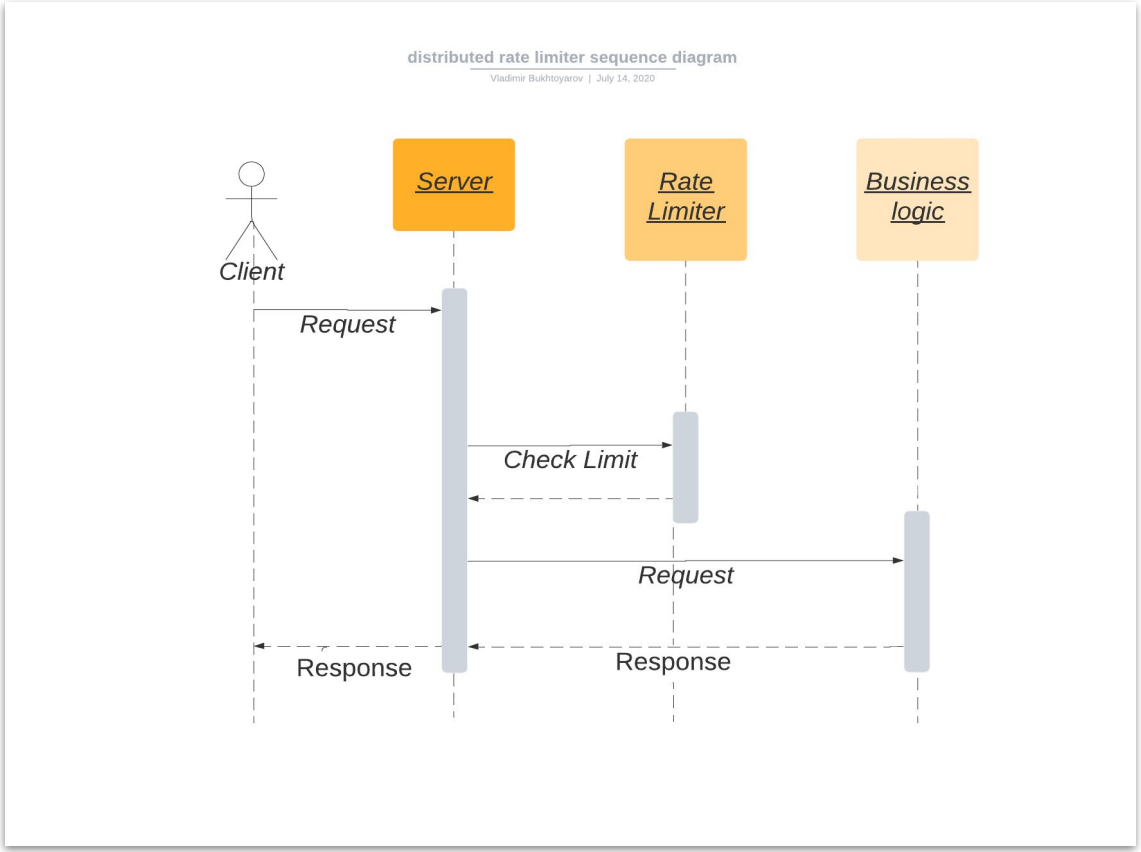
```java
@Override
public boolean tryAcquire(int permits) {
    State newState = new State();
    while (true) {
        long now = System.nanoTime();
        State previousState = stateReference.get();
        newState.availableTokens = previousState.availableTokens;
        newState.lastRefillNanotime = previousState.lastRefillNanotime;
        refill(newState, now);
        if (newState.availableTokens < permits) {
            return false;
        }
        newState.availableTokens -= permits;
        if (stateReference.compareAndSet(previousState, newState)) {
            return true;
        }
    }
}

private void refill(State state, long now) {
    long nanosSinceLastRefill = now - state.lastRefillNanotime;
    if (nanosSinceLastRefill <= nanosToGenerateToken) {
        return;
    }
    long tokensSinceLastRefill = nanosSinceLastRefill / nanosToGenerateToken;
    state.availableTokens = Math.min(capacity, state.availableTokens + tokensSinceLastRefill);
    state.lastRefillNanotime += tokensSinceLastRefill * nanosToGenerateToken;
}
```

Реализуем распределенный Token Bucket

- Выбор технологии.
- Пример кода.
- Обзор проблем.

Проблема RateLimiter в кластере:





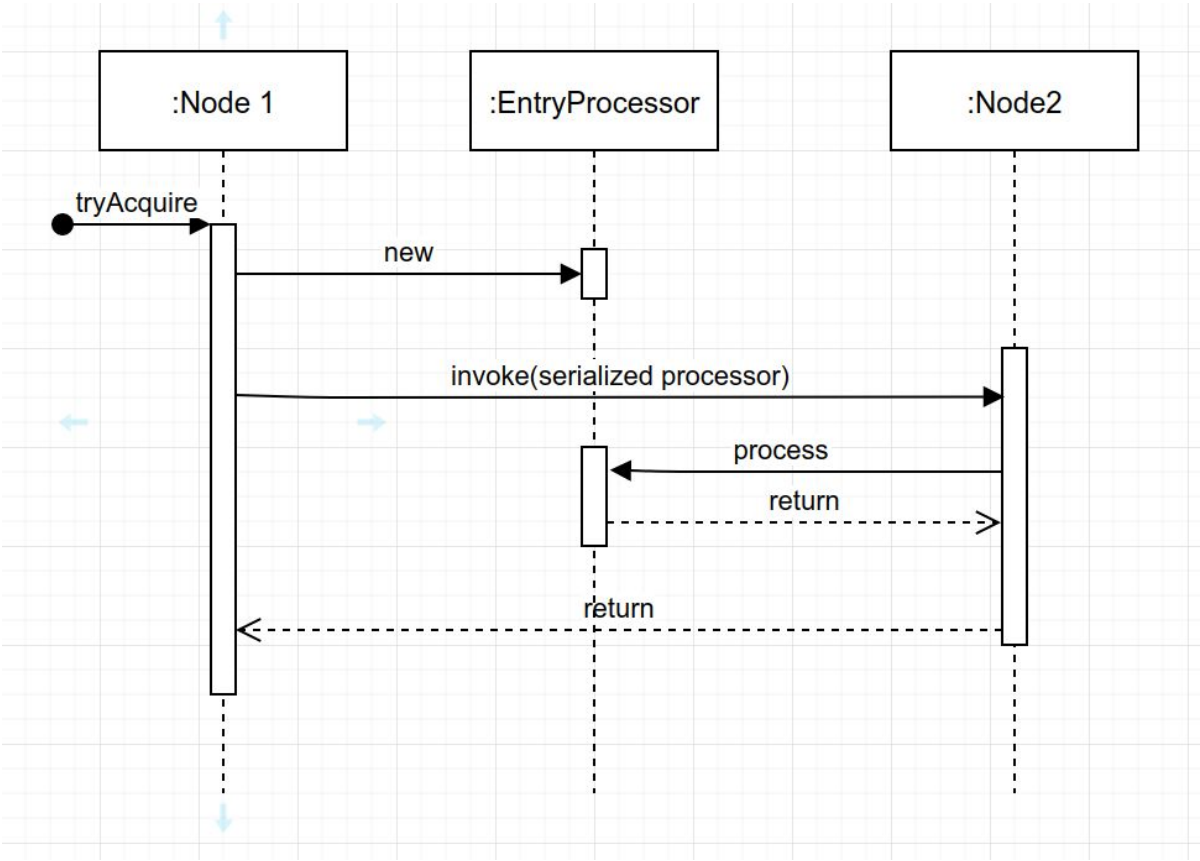
Требования к RateLimiter в кластере:

- Работать быстро. Желательно укладываться в 1 миллисекунду.
- Корректно работать в условиях параллельного доступа.
- Решение должно основываться на известных проверенных инструментах.

Пути реализации RateLimiter в кластере:

- DBMS + Select for Update?
  Всегда 3 сетевых запроса.
- MemCached, MongoDB,... + Compare And SWAP?!
  Чаще всего 2 запроса, однако в случае высокой конкуренции число CAS циклов не известно.
- DBMS + Stored procedures?!
  Всегда 1 сетевой запрос.
- Redis, Tarantool, Aerospike + Stored procedures!
  Всегда 1 сетевой запрос.
- JSR 107(Hazelcast, Coherence, Ignite) + EntryProcessor!!!
  Всегда 1 сетевой запрос. Не требует дополнительных серверов!

EntryProcessor крупным планом:



Рецепт достижения low-latency:

Направляйте вычисления к месту хранения данных.

Вместо того, чтобы подтягивать данные к месту проведения вычислений.

EntryProcessor изоляция транзакций:

- Изоляция транзакций в подавляющем большинстве случаев обеспечивается GRID-ом (исключение JBoss Infinispan).
- Изоляцию можно случайно отключить, поэтому всегда лучше тестировать
  https://github.com/vladimir-bukhtoyarov/bucket4j/blob/4.10/doc-pages/production-jcache-checklist.md#verification-of-compatibility-with-particular-jcache-provider-is-your-responsibility

Interface EntryProcessor<K,V,T>:

Type Parameters:

K - the type of keys maintained by this cache

V - the type of cached values

T - the type of the return value

`public interface` **`EntryProcessor<K,V,T>`**

An invocable function that allows applications to perform compound operations on a `Cache.Entry` atomically, according the defined consistency of a `Cache`.

Any `Cache.Entry` mutations will not take effect until after the `process(MutableEntry, Object...)` method has completed execution.

If an exception is thrown by an `EntryProcessor`, a Caching Implementation must wrap any `Exception` thrown wrapped in an `EntryProcessorException`. If this occurs no mutations will be made to the `Cache.Entry`.

Implementations may execute `EntryProcessors` in situ, thus avoiding locking, round-trips and expensive network transfers.

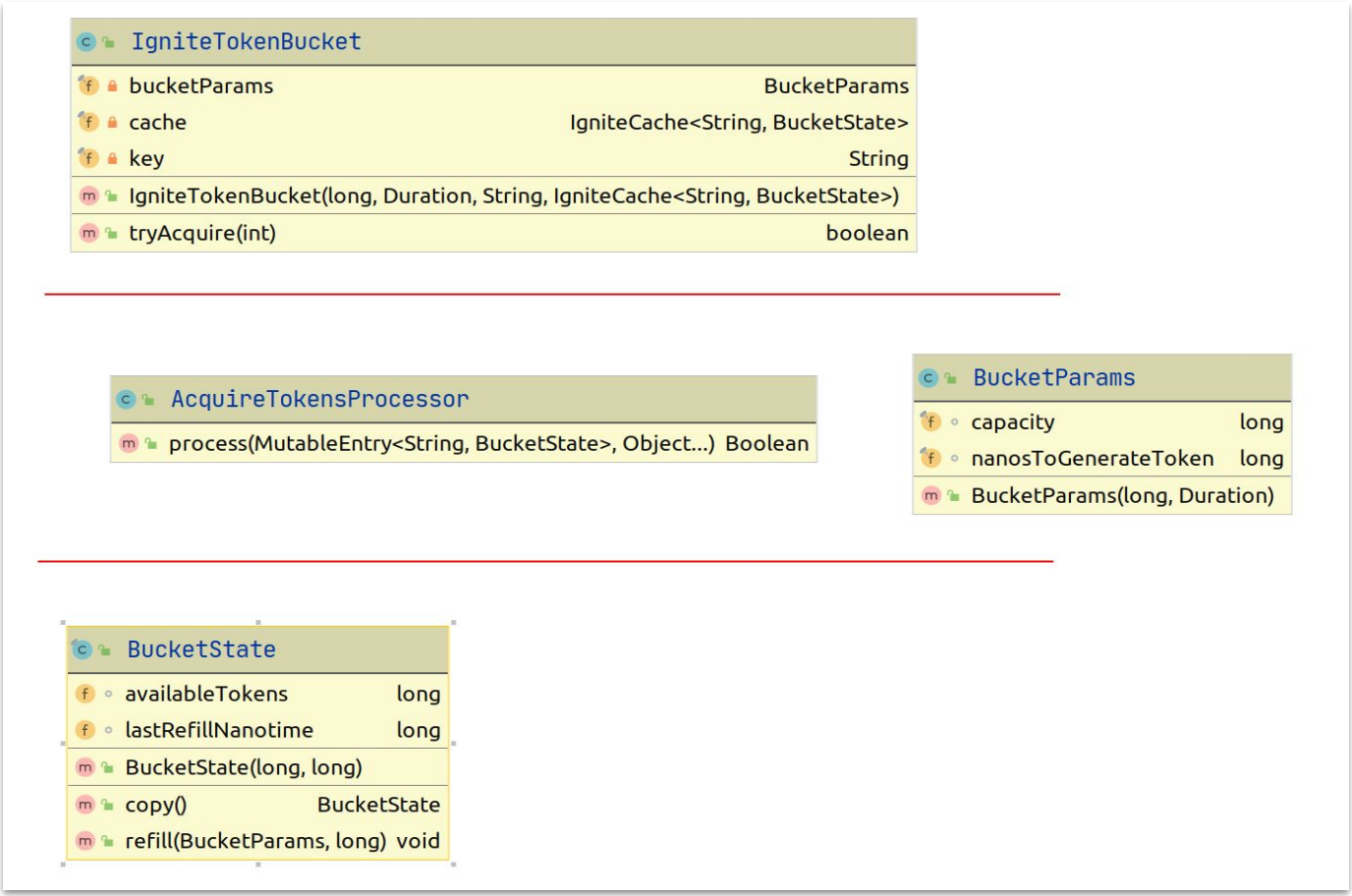
Do not use Redisson, and similar JCache adapters

Какую реализацию JSR-107 выбрать

- Hazelcast?
- Ignite?
- Coherence?

Да практически любую, **invoke(processor)** - это базовый функционал, примерно одинаково работающий в топовых реализациях.

IgniteTokenBucket диаграмма классов



DINS

IgniteTokenBucket состояние хранимое в кэше:

```
public final class BucketState implements Serializable {

    long availableTokens;
    long lastRefillNanotime;

    public BucketState(long availableTokens, long lastRefillNanotime) {
        this.availableTokens = availableTokens;
        this.lastRefillNanotime = lastRefillNanotime;
    }

    public BucketState copy() {
        return new BucketState(availableTokens, lastRefillNanotime);
    }

    public void refill(BucketParams params, long now) {
        long nanosSinceLastRefill = now - lastRefillNanotime;
        if (nanosSinceLastRefill <= params.nanosToGenerateToken) {
            return;
        }
        long tokensSinceLastRefill = nanosSinceLastRefill / params.nanosToGenerateToken;
        availableTokens = Math.min(params.capacity, availableTokens + tokensSinceLastRefill);
        lastRefillNanotime += tokensSinceLastRefill * params.nanosToGenerateToken;
    }

}
```

IgniteTokenBucket параметр для EntryProcessor:

```
public class BucketParams implements Serializable {

    final long capacity;
    final long nanosToGenerateToken;

    public BucketParams(long capacity, Duration period) {
        this.capacity = capacity;
        this.nanosToGenerateToken = period.toNanos() / capacity;
    }

}
```

IgniteTokenBucket реализация EntryProcessor:

```
public class AcquireTokensProcessor implements Serializable,
        EntryProcessor<String, BucketState, Boolean> {

    @Override
    public Boolean process(MutableEntry<String, BucketState> entry,
           Object... arguments) throws EntryProcessorException {
        final int tokensToConsume = (int) arguments[0];
        final BucketParams params = (BucketParams) arguments[1];
        long now = System.currentTimeMillis() * 1_000_000L;

        BucketState state;
        if (!entry.exists()) {
            state = new BucketState(params.capacity, now);
        } else {
            BucketState persistedState = entry.getValue();
            state = persistedState.copy();
            state.refill(params, now);
        }
        if (state.availableTokens < tokensToConsume) {
            return false;
        }
        state.availableTokens -= tokensToConsume;
        entry.setValue(state);
        return true;
    }

}
```

IgniteTokenBucket local part:

```
public class IgniteTokenBucket implements RateLimiter {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    public IgniteTokenBucket(long permits, Duration period,
              String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    @Override
    public boolean tryAcquire(int permits) {
        AcquireTokensProcessor processor = new AcquireTokensProcessor();
        return cache.invoke(key,processor, permits, bucketParams);
    }

}
```

IgniteTokenBucket blocking IO problem:

```
public class IgniteTokenBucket implements RateLimiter {

    private final BucketParams bucketParams;
    private final IgniteCache<String, BucketState> cache;
    private final String key;

    public IgniteTokenBucket(long permits, Duration period,
             String key, IgniteCache<String, BucketState> cache) {
        this.bucketParams = new BucketParams(permits, period);
        this.key = key;
        this.cache = cache;
    }

    @Override
    public boolean tryAcquire(int permits) {
        AcquireTokensProcessor processor = new AcquireTokensProcessor();
        return cache.invoke(key, processor, permits, bucketParams);
    }

}
```

IgniteAsyncTokenBucket:

```
public CompletableFuture<Boolean> tryAcquire(int numberTokens) {
    AcquireTokensProcessor entryProcessor = new AcquireTokensProcessor();
    IgniteFuture<Boolean> igniteFuture = cache.invokeAsync(key, entryProcessor, numberTokens, bucketParams);
    return convertFuture(igniteFuture);
}

private static <T> CompletableFuture<T> convertFuture(IgniteFuture<T> igniteFuture) {
    CompletableFuture<T> completableFuture = new CompletableFuture<>();
    igniteFuture.listen((IgniteInClosure<IgniteFuture<T>>) completedIgniteFuture -> {
        try {
            completableFuture.complete(completedIgniteFuture.get());
        } catch (Throwable t) {
            completableFuture.completeExceptionally(t);
        }
    });
    return completableFuture;
}
```

Итоги:

- Научились имплементировать Lock-Free Token Bucket.
- Научились избегать арифметические и performance ловушки.
- Научились имплементировать distributed Token Bucket с помощью Apache Ignite.

Полезные ссылки:

- https://en.wikipedia.org/wiki/Token_bucket
- https://github.com/bbeck/token-bucket
- https://github.com/vladimir-bukhtoyarov/bucket4j
- https://github.com/vladimir-bukhtoyarov/bucket4j/blob/4.10/doc-pages/production-jcache-checklist.md#verification-of-compatibility-with-particular-jcache-provider-is-your-responsibility
- https://github.com/vladimir-bukhtoyarov/bucket4j/blob/4.10/doc-pages/production-generic-checklist.md#be-wary-of-short-timed-bursts
- https://linkmeup.gitbook.io/sdsm/15.-qos/7.-ogranichenie-skorosti/4-mekhanizmy-leaky-bucket-i-token-bucket/1-algoritm-token-bucket

Thank you

Vladimir Bukhtoyarov

jsecoder@mail.ru

http://github.com/vladimir-bukhtoyarov/token-bucket-demo