Hazelcast

1. Tổng quan

1.1. Hazelcast là gì ?

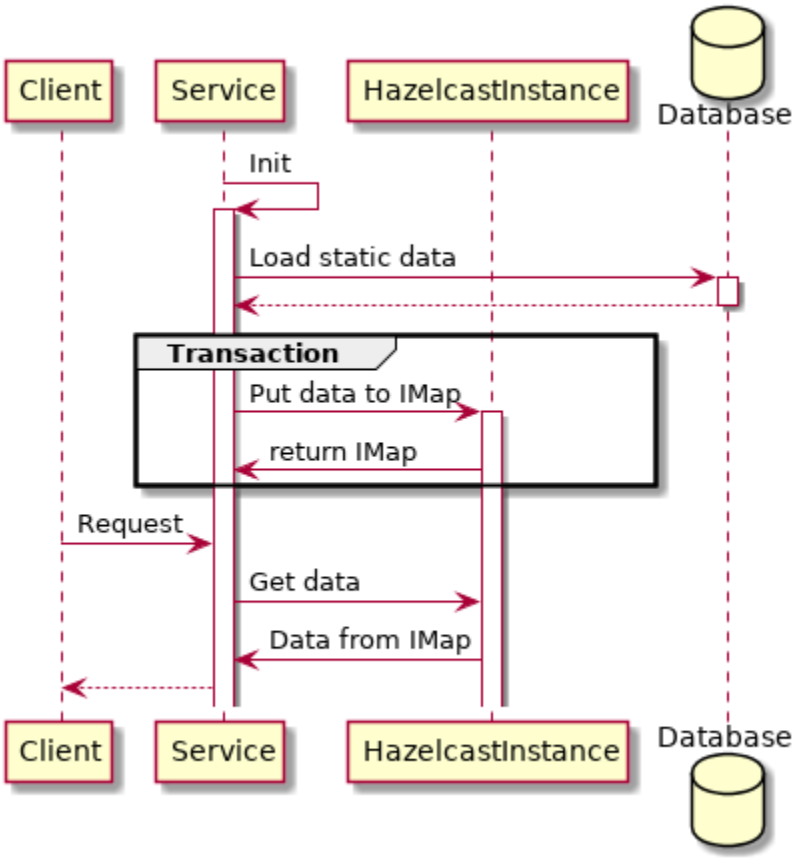
> *Mạng lưới các node lưu trữ dữ liệu phân tán dưới dạng in-memory*

- *"In-Memory Data Grid"* : Hazelcast được biết đến là 1 hệ thống lưu trữ dữ liệu phân tán giữa nhiều node (nhiều server), các node này kết hợp với nhau tạo thành 1 mạng lưới (grid) chia sẻ dữ liệu. Các node này hoạt động ngang hàng với nhau theo hình thức Peer-to-peer, không có node Master & Slave

=> Dữ liệu có thể lưu trữ ở node A, sau đó lấy ra từ node B hoặc bất cứ node nào trong grid.

- Dữ liệu được lưu trữ hoàn toàn trên RAM (In-Memory) dưới dạng không quan hệ (non-relational) và object-based. Ở phía Client, các dữ liệu này được wrapper lại bằng các interface Java (Map, List...) để lập trình viên dễ dàng thao tác.

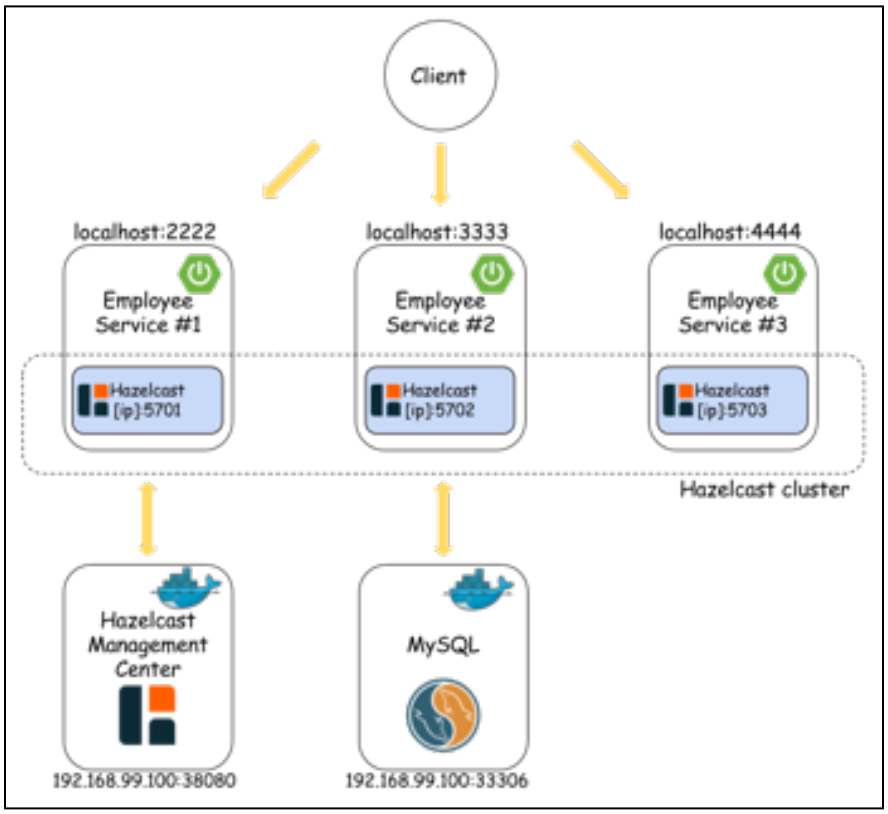
* *Cách sử dụng thông thường của team*:



- Service được khởi động, dữ liệu static được load từ database (các giá trị config system, business...)
- Các dữ liệu này sau đó được lưu vào Map của Hazelcast.
- Các request từ Client đến Service có sử dụng dữ liệu trên sẽ được lấy trực tiếp từ Hazelcast thay vì lấy từ database.
- Ngoài ra, Service thường cung cấp 1 api để reload lại Map này khi có sự thay đổi giá trị trong database.

- Hazelcast thường được so sánh với Redis về khía cạnh mục đích sử dụng nhưng khác nhau về bản chất lưu trữ. Hazelcast là *"In-Memory Data Grid"* trong khi Redis là *"In-Memory Database"*. Ngoài ra cũng nên lưu ý Redisson (1 thư viện wrapper của Redis mà ta hay sử dụng) lại được xem là *"In-Memory Data Grid"* nên giữa Hazelcast và Redisson có khá nhiều điểm tương đồng.

(Redisson definition: Redis Java Client with features of In-Memory Data Grid)



(Hình 1. Các node Hazelcast tạo thành mạng lưới lưu dữ liệu giữa 3 instance của EmployeeService)

1.2. Vì sao nên dùng Hazelcast ?

- Nhanh: dữ liệu lưu hoàn toàn trên RAM nên việc truy xuất chắc chắn nhanh hơn lấy dữ liệu trực tiếp từ database.
- Giảm tải cho database: mục đích chính của Hazelcast thường là một secondary datastore thay cho database; Dữ liệu từ database được load vào Hazelcast để các service sử dụng thay vì lấy trực tiếp ở database với mỗi request, giúp giảm tải số lượng request hit vào database.
- Dễ dàng mở rộng: Hazelcast được thiết kế để scale up lên đến hàng trăm, hàng ngàn node. Chỉ cần thêm node mới, Hazelcast sẽ tự động kết nối vào mạng lưới đã có để mở rộng cả bộ nhớ lẫn khả năng xử lý.
- Không có điểm chết (No single point of failure): Vì mạng lưới không có node Master, nên việc 1 node bị lỗi hoàn toàn không ảnh hưởng đến hoạt động của các node còn lại cũng như toàn bộ mạng lưới.

2. Mô hình triển khai

- Hazelcast hỗ trợ 2 cơ chế triển khai cluster bao gồm: Embedded Mode & Client-Server Mode.

(Hình 2. Các mô hình triển khai hệ thống các node của Hazelcast)

2.1. Embedded Mode:

- Đây là mode thường hay được sử dụng (và cũng là recommend của Hazelcast khi nhu cầu của bạn là truy xuất dữ liệu nhanh).
- Ở mode này, Hazelcast được "nhúng vào" (embedded) chính service đang chạy như 1 external lib và sử dụng 1 port riêng để giao tiếp với các node khác trong grid. Để sử dụng mode này, chỉ cần thêm lib Hazelcast vào file pom.xml của service.

```
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast</artifactId>
    <version>3.12.9</version>
</dependency>
```

- Dữ liệu trong mode này truy xuất nhanh hơn vì đang chứa trong chính bản thân node.
- Cần lưu ý rằng bộ nhớ cung cấp cho Hazelcast lúc này cũng chính là bộ nhớ heap của service đang chạy => Có thể sinh ra bottleneck nếu các node chạy với cấu hình vật lý khác nhau (ví dụ: 1 node 1Gb RAM, 1 node 8Gb, khi lưu trữ & truy xuất dữ liệu nhiều vào node 1Gb RAM có thể giảm hiệu năng cho toàn bộ cluster).

2.2. Client-Server Mode:

- Ở mode này, các node của cluster được triển khai độc lập, tương tự như cluster của Redis. Các ứng dụng Client sử dụng lib Hazelcast Client để kết nối đến cụm cluster.

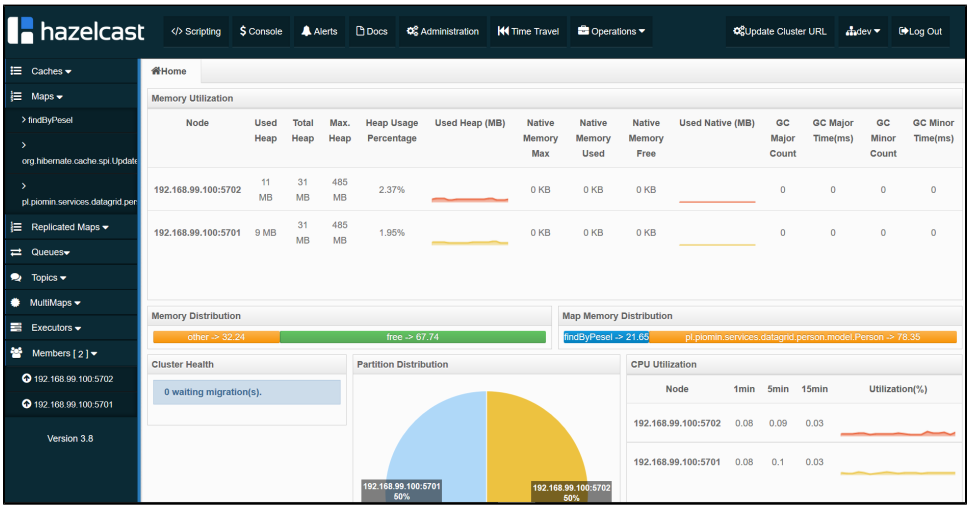
```
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast-client</artifactId>
    <version>3.12.9</version>
</dependency>
```

- Nhờ cơ chế triển khai độc lập, cụm cluster sẽ dễ mở rộng hơn, chỉ cần add thêm node Hazelcast khi cần mà không cần chạy thêm 1 instance của service, tách biệt infrastructure và business service. Đồng thời, cụm cluster trên có thể sử dụng cùng lúc cho nhiều nhóm service khác nhau.

- Bên cạnh đó cũng có các hạn chế như tăng độ phức tạp trong việc quản lý hạ tầng, tăng latency khi từ ứng dụng phải kết nối đến cụm cluster trên để thao tác với dữ liệu => Chỉ nên sử dụng khi có nhu cầu tách biệt kiến trúc & mở rộng số node Hazelcast.

2.3. Management Center

- Bên cạnh 2 mode triển khai như trên, Hazelcast còn hỗ trợ trang dashboard gọi là Management Center giúp quản lý chi tiết về cụm cluster, thông số các instance như memory, uptime... cũng như thao tác dừng/ ngắt hoạt động của các node trong cluster. Chi tiết cách cài đặt & các tính năng hỗ trợ tham khảo tại đây: https://docs.hazelcast.org/docs/management-center/latest/manual/html/index.html



(Hình 3. Giao diện Hazelcast Management Center)

3. Các kiểu dữ liệu hỗ trợ

* Hazelcast hỗ trợ hầu hết các kiểu dữ liệu cơ bản mà ta thường thao tác trong Java, các interface này được wrapper lại thành các cài đặt như sau:

- *Map*: implement của java.util.Map, bao gồm nhiều cài đặt cụ thể như *IMap* **(kiểu dữ liệu quan trọng nhất của Hazelcast)**, *TransactionalMap* (map có hỗ trợ Transaction), *MultiMap* (map với 1 key có thể chứa collection của value), *ReplicatedMap* (map được replicate trên nhiều node) (chi tiết sẽ được đề cập đến ở phần 4 & 5).

- *List, Set*: implement của java.util.List & Set, cũng có các cài đặt cụ thể như *IList*, *TransactionalList*, *ISet*, *TransactionalSet*.

- *Topic, Queue*: Các kiểu dữ liệu hoạt động tương tự như 1 message broker/ message queue để gửi & nhận thông điệp, xử lý background job.

- *Lock, ISemaphore, ICountdownLack*: các cài đặt của các interface trong package *java.util.concurrent*, hỗ trợ xử lý đụng độ ở cấp độ cluster - chỉ có 1 thread trên toàn cluster được sử dụng resource.

* Từ các kiểu dữ liệu trên, Hazelcast chia thành 2 nhóm chính:

- Nhóm kiểu dữ liệu có phân chia partition - dữ liệu của nhóm này được chia đều trên các node của cluster, bao gồm *Map*, *Multimap*.

- Nhóm kiểu dữ liệu không chia partition - toàn bộ dữ liệu của collection nằm trên 1 node, bao gồm *List, Set, Topic, Queue, Lock*...Vì đặc tính không phân chia dữ liệu, ta nên hạn chế sử dụng các kiểu dữ liệu này vì nó làm mất đi tính "distributed" của cluster, đồng thời có thể gây nghẽn node chứa dữ liệu đó.

Ví dụ: Đối với kiểu IQueue, nếu phía consumer không consume kịp so với tốc độ gửi message của producer, phía producer sẽ bị tràn bộ nhớ (OutOfMemoryError) do các message này đều nằm toàn bộ trên 1 node.

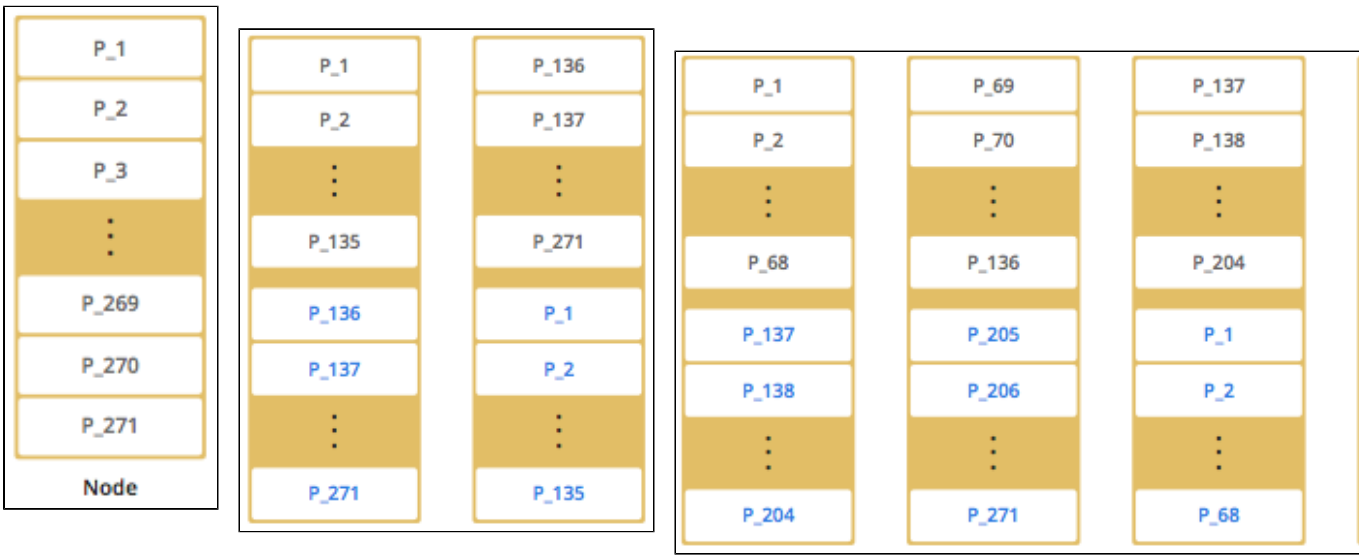
4. Phân bố dữ liệu

4.1. Cơ chế phân chia partition

- Như đã đề cập ở phần 3, Hazelcast thực hiện phân chia dữ liệu vào các partition trên mỗi node trong cluster để lưu trữ. Các partition này (mặc định là 271 partitions) được phân bố đều và ngẫu nhiên khắp các node trong cluster.

- Khi có sự thay đổi số lượng node trong cluster, quyền sở hữu các partitions này sẽ được phân chia lại. Node "già nhất" trong cluster (node start trước/ alive-time dài nhất) sẽ giữ thông tin quyền sở hữu partition trong 1 cấu trúc gọi là partition table, cho biết tình trạng phân bố các partition ở các node để có thể thực hiện rebalance khi có change. Đồng thời, thông tin này cũng định kỳ được sync đến các node còn lại để

phòng trường hợp node "già nhất" ngủm :v khi đó, node có "tuổi đời" cao nhất tiếp theo sẽ "kế nhiệm" công việc rebalance.



(Hình 4. Phân bố các partition khi số lượng node trong cluster thay đổi)

- Tại thời điểm cluster chỉ có 1 node, toàn bộ 271 partitions đều nằm bên trong node này. Sau đó khi tăng thêm 1 node, phân nửa số partition đã được phân chia sang node 2 (các partition được chọn ngẫu nhiên). Đồng thời, mỗi partition của node này sẽ có 1 bản backup nằm ở node còn lại, nhờ vào bản backup này mà Hazelcast có thể phục hồi lại dữ liệu khi có node down. Tương tự cho trường hợp số node tăng lên 4 => số partition được chia đều làm 4 và cũng có tương ứng số bản backup.

4.2. Cơ chế phân bố dữ liệu

Hazelcast thực hiện phân chia dữ liệu vào các partition sử dụng 1 thuật toán hashing như sau:

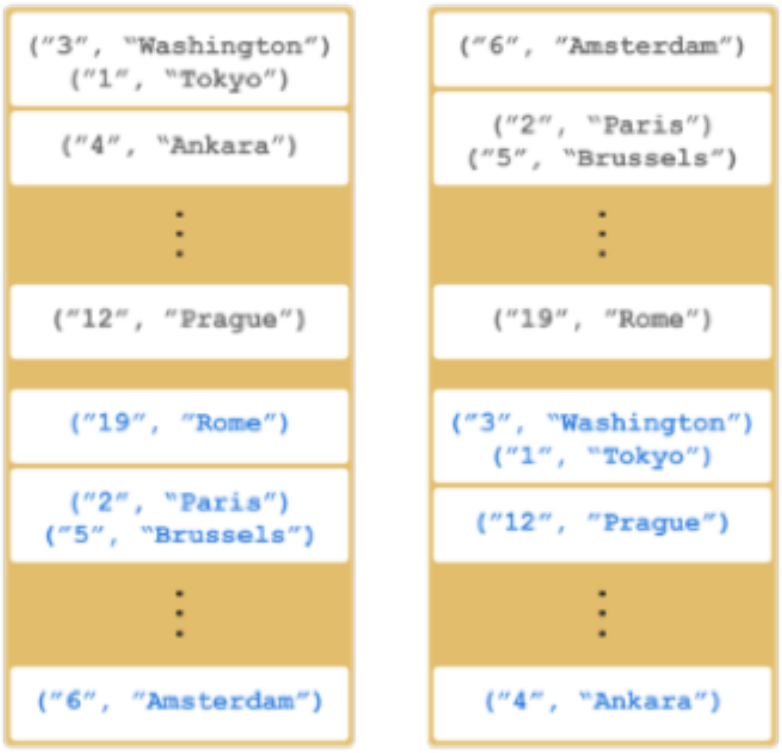
- Serialize key của object thành 1 mảng byte.
- Mảng byte này được hash sử dụng Murmur Hash 3 (cài đặt chi tiết: https://github.com/hazelcast/hazelcast/blob/master/hazelcast/src/main/java/com/hazelcast/internal/util/HashUtil.java)
- Lấy kết quả của phép hash trên mod cho số lượng partition.
- Kết quả cuối: MOD(hash, partition count) - chính là partitionID mà dữ liệu sẽ được lưu trữ.

**Để hiểu rõ hơn quá trình phân chia partition của Hazelcast có thể xem video sau:*

5. IMap

5.1. Tổng quan

- Kiểu dữ liệu đặc trưng của Hazelcast, được khởi tạo khi gọi method *hazelcastInstace.getMap()* nếu chưa tồn tại hoặc trả về map trước đó => map này không bao giờ null. Đồng thời, IMap kế thừa từ *ConcurrentMap* nên bản thân nó cũng là thread-safe.
- Dữ liệu (primary & backup) được phân chia khắp các node dựa trên key, mỗi entry của map thuộc về 1 partition (cách tính partitionId tham khảo section 4).



(Hình 5. Phân bố các entry của map tại các partition)

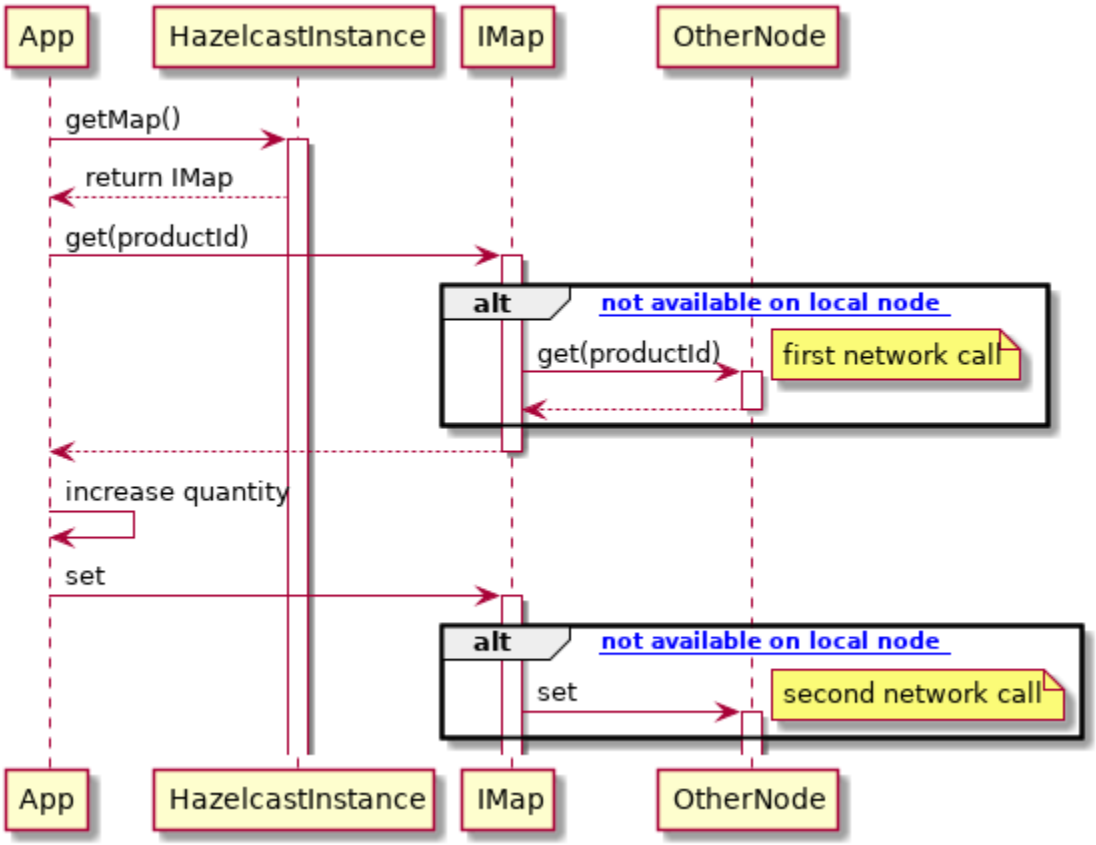
- IMap, ngoài các method đặc trưng của Map Java, còn hỗ trợ rất nhiều tính năng khác để thao tác như:
  + MapEviction: Cơ chế tự động xóa các phần tử của map dựa trên policy như LRU, LFU, số lượng phần tử...thường được dùng khi map đóng vai trò cache.
  + MapLoader: Cơ chế liên kết giữa map và external datasource như MySQL, MongoDB,...bất cứ khi nào get 1 entry không tồn tại trong map, Hazelcast sẽ thực hiện query dữ liệu từ db để put vào map, sau đó trả kết quả về cho client; tương tự, khi thực hiện put dữ liệu vào map, Hazelcast thực hiện query insert xuống datasource tương ứng.
  + EntryListener: Cơ chế thông báo sự kiện khi có sự thay đổi dữ liệu trong map bao gồm thêm, xóa phần tử, cập nhật giá trị... Service có thể lắng nghe các sự kiện này để xử lý tương ứng.
  + EntryProcessor, Near Cache: (xem bên dưới)

5.2. Entry Processor

- Xem xét đoạn code sau:

```
IMap<Long, Integer> cart = hazelcastInstance.getMap("cart");
int quantity = cart.get(productId);
int newQuantity = quantity + 1;
cart.set(productId, newQuantity);
```

- Sequence diagram của đoạn code trên:

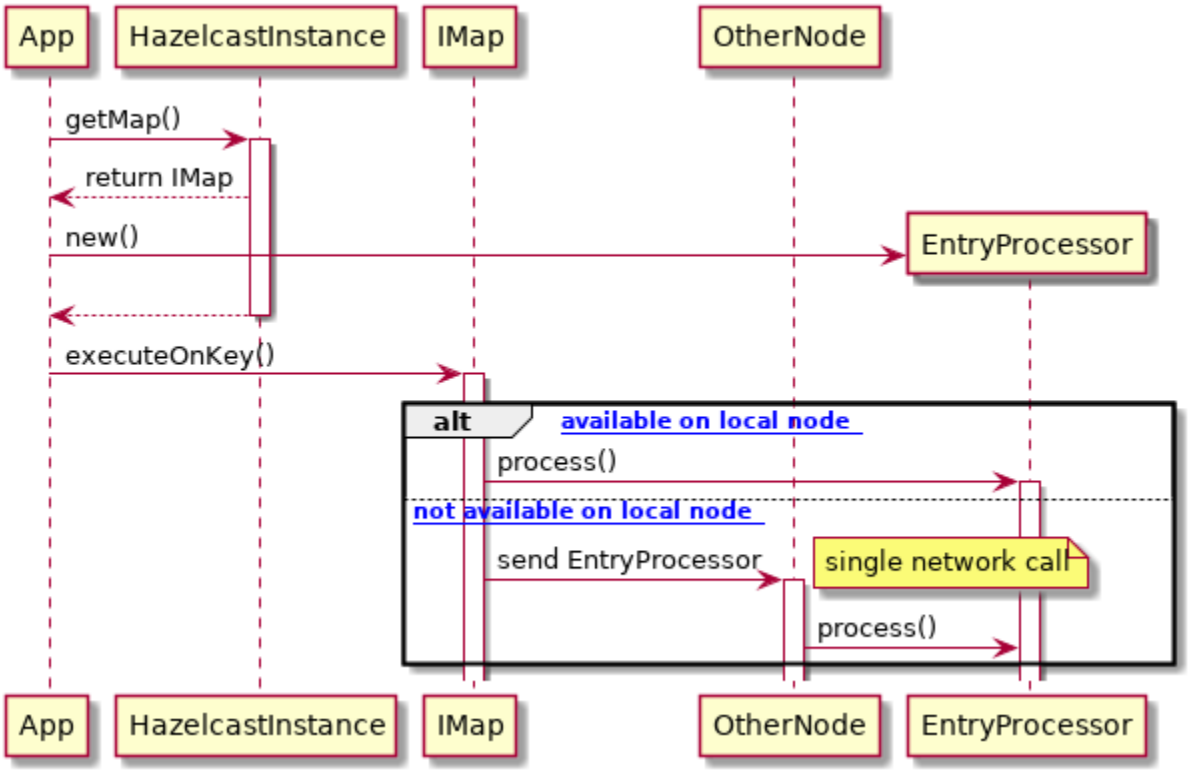


- Như đã biết, mỗi entry của map nằm ở 1 partition xác định, và có thể không nằm ở node mà client đang gửi command tới. Lúc này, node đó phải chuyển tiếp command đến node tương ứng chứa entry để xử lý, như trong hình ta có 2 lần mà node hiện tại phải truy vấn đến node chứa entry, tốn 2 network call. Đây cũng chính là 1 trong những nguyên nhân quan trọng nhất gây giảm performance của Hazelcast, khi liên tục phải truy vấn đến các node không chứa dữ liệu tương ứng (giống với cache miss).

- Để khắc phục vấn đề trên, Hazelcast đưa ra 1 giải pháp như sau: thay vì lấy phần tử ra để truyền vào method/xử lý thì hãy gửi method đến nơi đang chứa phần tử đó để thao tác, thông qua EntryProcessor. Code ví dụ như sau:

```
IMap<Long, Integer> cart = hazelcastInstance.getMap("cart");
int newQuantity = (int) cart.executeOnKey(productId, new AbstractEntryProcessor<Long, Integer>() {
            @Override
            public Object process(Map.Entry<Long, Integer> entry) {
                int quantity = entry.getValue();
                int newQuantity = quantity + 1;
                entry.setValue(newQuantity);
                return newQuantity;
            }
        });
```

- Ta có thể thấy 1 method implement AbstractEntryProcessor được "gửi đến" nơi chứa productId cần xử lý thông qua việc gọi executeOnKey. Sequence tương ứng như sau (lưu ý chỉ tốn 1 network call duy nhất):



5.3. Near Cache

- Để giải quyết bài toán phải get entry từ các node khác nhiều lần, có thể sử dụng 1 trong 2 cách sau ddaya:

+ ReplicatedMap : 1 kiểu dữ liệu gần tương đương với 1 IMap, nhưng dữ liệu của toàn bộ map được chứa đầy đủ trên từng node, không có bản backup => mỗi node chứa bản copy đầy đủ của map nên không cần truy vấn đến node khác để lấy dữ liệu. Tuy nhiên cách này có hạn chế là ReplicatedMap không hỗ trợ transaction (chi tiết ở section 6).

+ Near-Cache : Cơ chế tạo 1 bản cache tại node bị miss dữ liệu, các request truy vấn cùng 1 key sau đó sẽ lấy từ bản cache này giúp tăng tốc độ xử lý. Hạn chế của cách này bao gồm việc phải cài đặt cache riêng cho từng map, cũng như việc data bị out-of-date (có thể khắc phục bằng cách set TTL ngắn ~60s). Thực hiện đo lường thực tế cho thấy trong trường hợp cache hit nhiều, performance có thể tăng lên gấp đôi, nhưng nếu trường hợp cache miss nhiều thì performance thực tế bị giảm do tốn thời gian cho việc tạo bản cache.

5.4. Chú ý

- Xem xét các đoạn code sau:

class BankConfig

```
public class BankConfig implements Serializable {

    private String bankCode;
    private int status;
}
```

Java Map

```
Map<String, BankConfig> bankMap = new HashMap<>();
bankMap.put("VCB", new BankConfig());

BankConfig bank = bankMap.get("VCB");
bank.setStatus(1);
log.info("Java Map : " + StringUtil.toJsonString(bankMap));
```

Hazelcast Map

```
Map<String, BankConfig> bankMap2 = hazelcastInstance.getMap("bankMap");
bankMap2.put("VCB", new BankConfig());

BankConfig bank2 = bankMap2.get("VCB");
bank2.setStatus(1);
log.info("Hazelcast Map : " + StringUtil.toJsonString(bankMap2));
```

- Đoạn code thứ 2 thực hiện thao tác trên 1 Java Map, thêm 1 entry có Key="VCB", value là object BankConfig và setStatus=1
- Đoạn code thứ 3 thực hiện tương tự, nhưng là thao tác trên Hazelcast IMap
- Kết quả in ra :

```
Java Map : {"VCB":{"status":1}}
Hazelcast Map : {"VCB":{"status":0}}
```

** Cùng 1 đoạn code xử lý nhưng lại cho ra 2 kết quả khác nhau, vậy nguyên nhân là do đâu ?*

- Đối với Hazelcast, khi thực hiện put 1 entry vào map, đã xảy ra 1 quá trình serialize dữ liệu sang dạng byte array để lưu trữ. Khi thực hiện get trở lại entry đó, quá trình deserialize từ byte array để "tái tạo" lại object đó dựa trên các constructor của class (chi tiết về serialize tham khảo section 7). Lúc này, object bạn nhận được là 1 bản clone/ copy được tạo ra từ dữ liệu của object gốc, không phải bản thân object đã được lưu trữ vào map ban đầu. Vậy nên việc thao tác cập nhật dữ liệu 1 entry là đang thao tác với bản copy đó, còn dữ liệu gốc không bị ảnh hưởng, chính vì vậy khi ta get ra entry đó từ map 1 lần nữa sẽ vẫn thấy dữ liệu cũ => muốn chỉnh sửa dữ liệu của object trong Hazelcast Map, sau khi cập nhật phải thực hiện put/set/replace object trở lại map để Hazelcast serialize lại.

```
// !!! Must put updated object
bankMap2.put("VCB", bank);
log.info("Hazelcast Map : " + StringUtil.toJsonString(bankMap2));
```

6. Transaction

- Để đảm bảo tính toàn vẹn dữ liệu khi có update, việc hỗ trợ transaction là điều gần như bắt buộc. Tuy nhiên, trong Hazelcast chỉ có 1 số các kiểu dữ liệu đặc trưng được hỗ trợ transaction bao gồm: *TransactionalMap, TransactionalList, TransactionalSet, TransactionalQueue*. Mỗi kiểu dữ liệu trên là 1 phiên bản mở rộng của kiểu dữ liệu gốc tương ứng, ví dụ: *TransactionalMap* bản chất là 1 *IMap*.

- Cách sử dụng cơ bản:

```
public class Transactional {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        TransactionOptions txOption =(new TransactionOptions())
                                                                    .setTimeout(60L, TimeUnit.SECONDS)
                                                .setTransactionType(TransactionOptions.TransactionType.
TWO_PHASE);
        TransactionContext txContext = hz.newTransactionContext(txOption);

        txContext.beginTransaction();
        try {
            TransactionalMap<Integer, Integer> map = txContext.getMap("map");
            map.put(1, 1);
            map.put(2, 2);
            txContext.commitTransaction();
        } catch (Exception ex) {
            txContext.rollbackTransaction();
        }
    }
}
```

+ Truy cập đến HazelcastInstance.

+ Cài đặt option cho transaction (timeout & commit mode) bằng class TransactionOption.

+ Khởi tạo 1 TransactionContext (context mà transaction này sẽ được thực thi) với các option trên.

+ Bắt đầu transaction với method context.beginTransaction().

+ Thực hiện thao tác trên các kiểu dữ liệu Transactional như TransactionalMap.

+ Commit transaction nếu thành công hoặc rollback nếu có lỗi xảy ra.

Lưu ý:

+ Các thao tác trên *TransactionalMap* như map.put sẽ lock key tương ứng đang được xử lý cho đến hết transaction (lock hành động cập nhật như put, replace, không lock get) nên cần giảm thiểu tối đa thời gian thực hiện transaction để tránh ảnh hưởng đến performance.

+ Nếu transaction bị rollback, chỉ có những thay đổi trên các kiểu dữ liệu Transactional bị rollback, các thay đổi trên các kiểu dữ liệu khác sẽ vẫn giữ nguyên.

TransactionOption: class option để cài đặt các tham số cho transaction bao gồm 2 thông số:

- Timeout: thời gian tối đa thực hiện transaction, vượt quá thời gian này transaction xem như thất bại và tự động rollback.

- Commit Mode: kiểu commit, gồm 2 kiểu sau

+ *ONE_PHASE* : chỉ có 1 phase là commit change, nếu có lỗi xảy ra (ví dụ: node commit bị crash trong lúc đang commit thì dữ liệu có thể bị inconsistence)

+ *TWO_PHASE* : bao gồm thêm 1 phase là prepare phase (phase chuẩn bị). Tại phase này, trước khi commit, node commit sẽ copy commit log của mình (nơi lưu trữ các change diễn ra trong transaction) sang các node trong cluster để đảm bảo khi có lỗi xảy ra trong quá trình commit, các node khác vẫn có đủ thông tin để tiếp tục hoàn thành commit.

** Vậy tại sao không chọn mode TWO_PHASE làm mặc định cho transaction?*

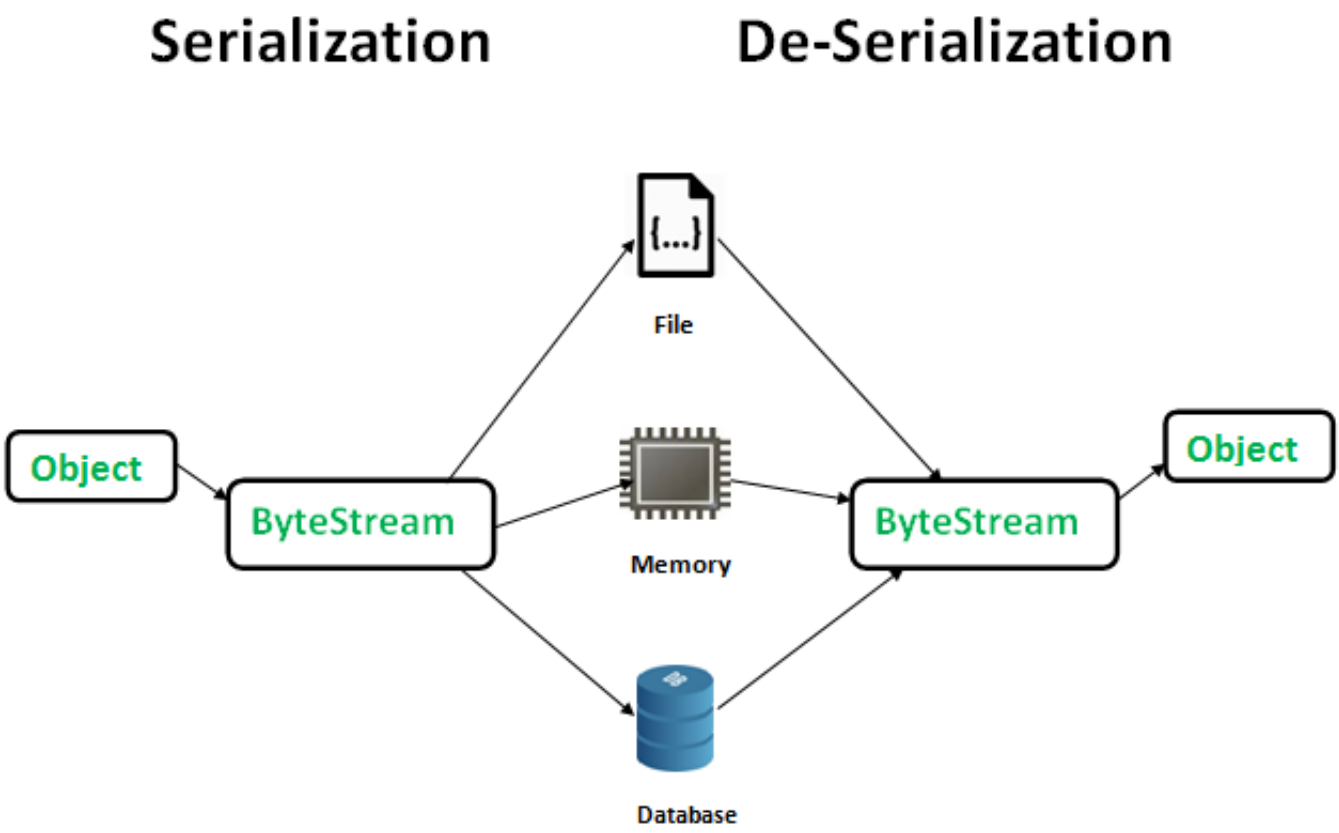
- Như đã nói phía trên, các thao tác trên *TransactionMap* trong transaction sẽ khiến key bị lock đến hết transaction, trong khi đó mode *TWO_PHASE* sẽ có thời gian thực hiện dài hơn do có bước copy commit log của prepare phase => kéo dài thời gian transaction (và có thể bị timeout) gây giảm performance. Nên việc quyết định sử dụng mode nào sẽ tùy thuộc vào nhu cầu, nếu muốn performance cao hãy chọn mode *ONE_PHASE*, còn mode *TWO_PHASE* sẽ cho tính consistence dữ liệu cao hơn.

7. Serialization

7.1. Serialization là gì ?

- Đối với các hệ thống xây dựng trên nền tảng JVM, khi trao đổi dữ liệu giữa các thành phần khác nhau (giữa các module cùng viết bằng Java, ghi dữ liệu ra file...) thì dữ liệu được thể hiện dưới dạng byte chứ không phải object. Do đó ta cần có một cơ chế để hiểu các object

được gửi và nhận. Quá trình đó được gọi là "**serialization**" - chuyển đổi 1 object thành dạng binary, và quá trình ngược lại được gọi là "**deserialization**".



(Hình 6. Serialization là gì ?)

- Trong Hazelcast, quá trình serialization xảy ra trong các trường hợp sau đây:

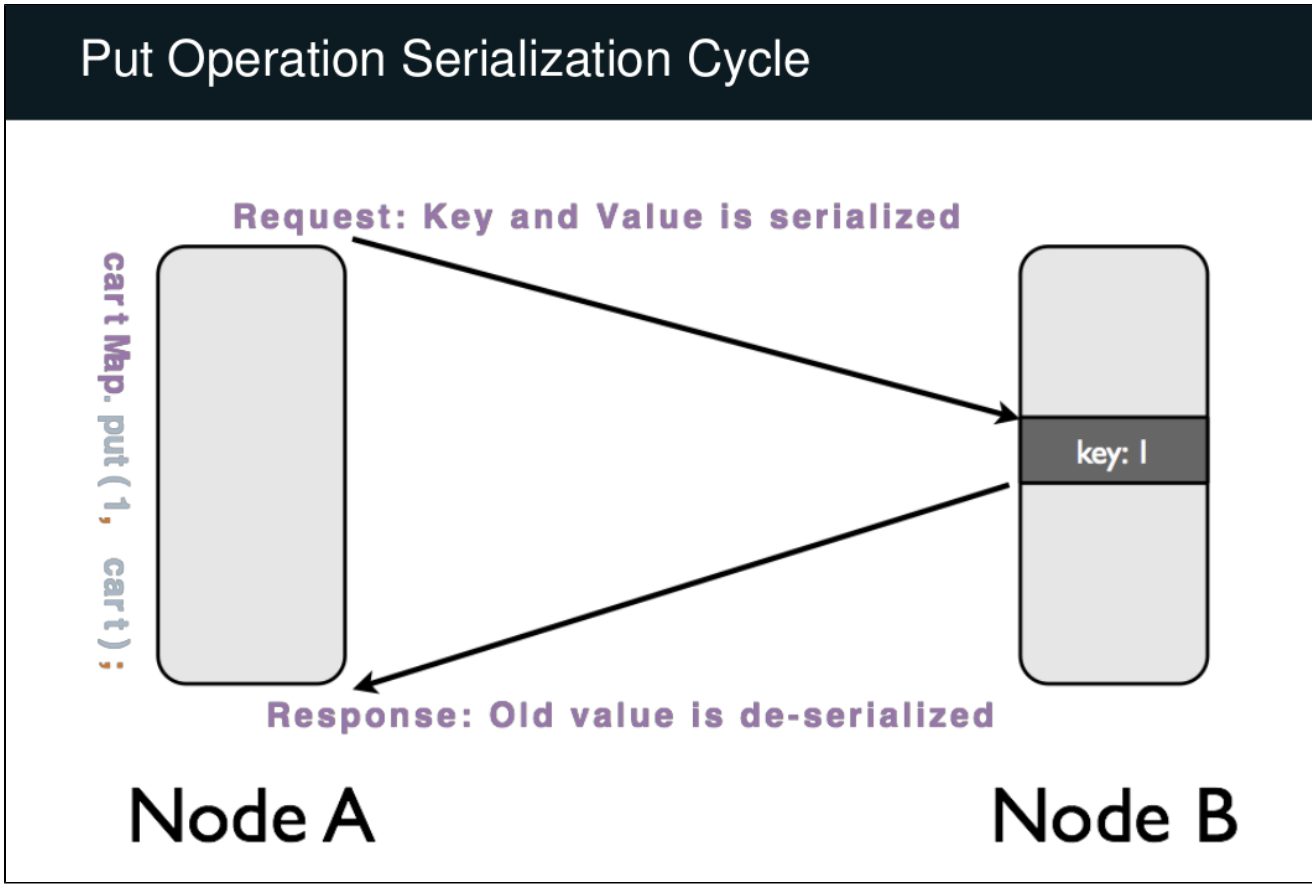
  - Thêm cặp Key/Value vào IMap, hoặc xử lý phần tử trong Map.

  - Thêm phần tử mới vào IQueue, ISet, IList.

  - Lock 1 phần tử (1 key).

  - Gửi message vào 1 topic.

  - ...



(Hình 7. Quá trình serialize khi gọi method map.put() của Hazelcast)

Lưu ý: Method **map.put** trả về phần tử mới sau khi thêm vào map, việc này tốn thêm 1 bước de-serialize lại object gây giảm performance, nên nếu không cần sử dụng giá trị vừa được put vào, ta có thể sử dụng hàm **map.set** (không trả về giá trị mới).

7.2. Các interface hỗ trợ serialization & thứ tự ưu tiên

- Để Hazelcast có thể thực hiện serialize 1 object, class object đó phải implement ít nhất 1 trong các interface sau đây:

  - Các interface chuẩn của Java:

```
java.io.Serializable
java.io.Externalizable
```

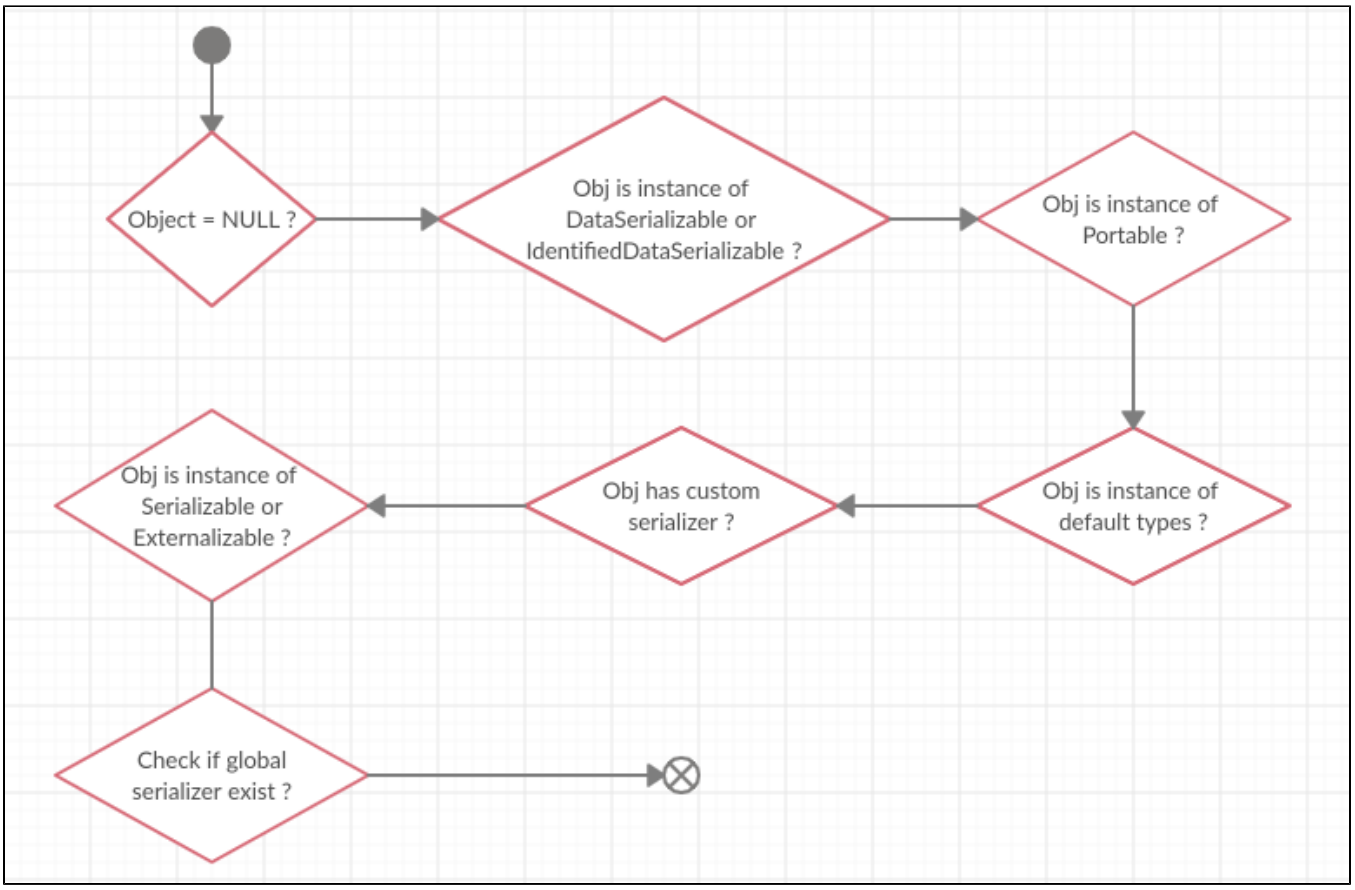
  - Các interface đặc trưng của Hazelcast:

```
com.hazelcast.nio.serialization.DataSerializable
com.hazelcast.nio.serialization.IdentifiedDataSerializable
com.hazelcast.nio.serialization.Portable
```

  - Các interface của Hazelcast hỗ trợ custom serialize với thư viện của bên thứ 3:

```
com.hazelcast.nio.serialization.ByteArraySerializer
com.hazelcast.nio.serialization.StreamSerializer
```

- Để tìm bộ cài đặt tương ứng để serialize object, Hazelcast thực hiện kiểm tra lần lượt các interface theo thứ tự dưới đây, và sẽ dừng kiểm tra & chọn bộ serializer tương ứng ngay khi thỏa điều kiện. Nếu sau khi kiểm tra tất cả các bước mà vẫn không tìm được bộ serializer tương ứng cho object thì Hazelcast sẽ báo exception.



(Hình 8. Thứ tự kiểm tra và tìm kiếm bộ serializer hợp lệ cho object của Hazelcast)

1. Kiểm tra object khác NULL (không thể serialize NULL)
2. Kiểm tra object có implement 2 interface **DataSerializable**/ **IdentifiedDataSerializable**.
3. Kiểm tra object có implement interface **Portable**.
4. Kiểm tra object thuộc về các kiểu dữ liệu cơ bản (int, long, String,...).
5. KIểm tra object có được cài đặt 1 bộ custom serializer.
6. Kiểm tra object có implement 2 interface **Serializable**/ **Externalizable**.
7. Kiểm tra có tồn tại 1 bộ serializer ở cấp global (cho tất cả các class)

7.3. Cài đặt chi tiết các interface & so sánh

7.3.1. Cài đặt chi tiết các interface

(Tạo class Order chứa các thông tin cơ bản của 1 order để minh họa)

A. Interface Serializable

- Interface cơ bản của Java, chỉ cần object class implement interface này là đã sử dụng được.

```
public class SerializableOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    private long orderId;
    private String item;
    private long orderTime;
    private String email;
    private Date updateTime;
    private SerializableDetailOrder detailOrder;
}
```

```
public class SerializableDetailOrder implements Serializable {

    private static final long serialVersionUID = 2L;

    private long amount;
    private String productCode;
}
```

B. Interface Externalizable

- Interface cơ bản của Java, cần implement và cài đặt chi tiết cách read/ write các field trong object. Lưu ý thứ tự khi read phải tương tự với thư tự khi write.

```
public class ExternalizableOrder implements Externalizable {

    private long   orderId;
    private String item;
    private long   orderTime;
    private String email;
    private Date   updateTime;
    private ExternalizableDetailOrder detailOrder;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeLong(orderId);
        out.writeUTF(item);
        out.writeLong(orderTime);
        out.writeUTF(email);
        out.writeLong(updateTime.getTime());
        detailOrder.writeExternal(out);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        this.orderId = in.readLong();
        this.item = in.readUTF();
        this.orderTime = in.readLong();
        this.email = in.readUTF();
        this.updateTime = new Date(in.readLong());
        this.detailOrder = new ExternalizableDetailOrder();
        this.detailOrder.readExternal(in);
    }
}
```

```
public class ExternalizableDetailOrder implements Externalizable {

    private long amount;
    private String productCode;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeLong(amount);
        out.writeUTF(productCode);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        this.amount = in.readLong();
        this.productCode = in.readUTF();
    }
}
```

C. Interface DataSerializable

- Interface đặc trưng của Hazelcast, tương tự với Externalizable, đồng thời có thêm 1 số cải tiến về mặt resource giúp tăng performance.

```
public class DataSerializableOrder implements DataSerializable {

    private long orderId;
    private String item;
    private long orderTime;
    private String email;
    private Date updateTime;
    private DataSerializableDetailOrder detailOrder;

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeLong(orderId);
        out.writeUTF(item);
        out.writeLong(orderTime);
        out.writeUTF(email);
        out.writeLong(updateTime.getTime());
        detailOrder.writeData(out);
    }

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        this.orderId = in.readLong();
        this.item = in.readUTF();
        this.orderTime = in.readLong();
        this.email = in.readUTF();
        this.updateTime = new Date(in.readLong());
        this.detailOrder = new DataSerializableDetailOrder();
        this.detailOrder.readData(in);
    }
}
```

```
public class DataSerializableDetailOrder implements DataSerializable {

    private long amount;
    private String productCode;

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeLong(amount);
        out.writeUTF(productCode);
    }

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        this.amount = in.readLong();
        this.productCode = in.readUTF();
    }
}
```

D. Interface IdentifiedDataSerializable

- Interface đặc trưng của Hazelcast, tương tự DataSerializable. Ngoài ra, việc kết hợp với Factory method để tạo ra object class, bỏ đi việc phụ thuộc vào Reflection khiến interface này nhanh hơn DataSerializable nhiều, và đây cũng là recommend của Hazelcast khi thực hiện Serialize.

```
public class IdentifiedDataSerializableOrder implements IdentifiedDataSerializable {

    private long orderId;
    private String item;
    private long orderTime;
    private String email;
    private Date updateTime;
    private IdentifiedDataSerializableDetailOrder detailOrder;

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeLong(orderId);
        out.writeUTF(item);
        out.writeLong(orderTime);
        out.writeUTF(email);
        out.writeLong(updateTime.getTime());
        detailOrder.writeData(out);
    }

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        this.orderId = in.readLong();
        this.item = in.readUTF();
        this.orderTime = in.readLong();
        this.email = in.readUTF();
        this.updateTime = new Date(in.readLong());
        this.detailOrder = new IdentifiedDataSerializableDetailOrder();
        this.detailOrder.readData(in);
    }

    @Override
    public int getFactoryId() {
        return 1;
    }

    @Override
    public int getId() {
        return 1;
    }
}
```

```
public class IdentifiedDataSerializableDetailOrder implements IdentifiedDataSerializable {

    private long amount;
    private String productCode;

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeLong(amount);
        out.writeUTF(productCode);
    }

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        this.amount = in.readLong();
        this.productCode = in.readUTF();
    }

    @Override
    public int getFactoryId() {
        return 1;
    }

    @Override
    public int getId() {
        return 2;
    }
}
```

```
public class IdentifiedDataSerializableOrderFactory implements DataSerializableFactory {

    @Override
    public IdentifiedDataSerializable create(int typeId) {
        if ( typeId == 1 ) {
            return new IdentifiedDataSerializableOrder();
        }
        if (typeId == 2) {
            return new IdentifiedDataSerializableDetailOrder();
        }

        return null;
    }
}
```

E. Interface Portable

- Interface hỗ trợ việc các cluster member có thể sử dụng nhiều phiên bản khác nhau của cùng 1 object class, thông qua method getClassVersion(), giúp việc rolling upgrade trở nên dễ dàng hơn khi các version đang chạy không bị ảnh hưởng bởi version mới được update.

```
public class PortableOrder implements VersionedPortable {
    private static final int classVersion = 2;
    private long orderId;
    private String item;
    private long orderTime;
    private String email;
    private Date updateTime;
    private PortableDetailOrder detailOrder;

    @Override
    public int getFactoryId() {
        return 1;
    }

    @Override
    public int getClassId() {
        return 1;
    }

    @Override
    public void writePortable(PortableWriter writer) throws IOException {
        writer.writeLong("1", orderId);
        writer.writeUTF("2", item);
        writer.writeLong("3", orderTime);
        writer.writeUTF("4", email);
        writer.writeLong("5", updateTime.getTime());
        writer.writePortable("6", detailOrder);
    }

    @Override
    public void readPortable(PortableReader reader) throws IOException {
        this.orderId = reader.readLong("1");
        this.item = reader.readUTF("2");
        this.orderTime = reader.readLong("3");
        this.email = reader.readUTF("4");
        this.updateTime = new Date(reader.readLong("5"));
        this.detailOrder = reader.readPortable("6");
    }

    @Override
    public int getClassVersion() {
        return classVersion;
    }
}
```

```
public class PortableDetailOrder implements VersionedPortable {

    private static final int classVersion = 1;

    private long amount;
    private String productCode;

    @Override
    public int getFactoryId() {
        return 1;
    }

    @Override
    public int getClassId() {
        return 2;
    }

    @Override
    public void writePortable(PortableWriter out) throws IOException {
        out.writeLong("1",amount);
        out.writeUTF("2", productCode);
    }

    @Override
    public void readPortable(PortableReader in) throws IOException {
        this.amount = in.readLong("1");
        this.productCode = in.readUTF("2");
    }

    @Override
    public int getClassVersion() {
        return classVersion;
    }
}
```

```
public class PortableOrderFactory implements PortableFactory {

    @Override
    public Portable create(int classId) {
        if (classId == 1) {
            return new PortableOrder();
        }
        if (classId == 2) {
            return new PortableDetailOrder();
        }
        return null;
    }
}
```

F. Interface StreamSerializer

- Interface hỗ trợ việc custom serialize, có thể kết hợp với thư viện của bên thứ 3 như Kryo, JSON,...(ví dụ dưới đây sử dụng Kryo), hữu dụng khi class object cần serialize không thể chỉnh sửa được (giả sử class từ external library), vì class object không cần implement interface nào cả, chỉ cần cài đặt cách serialize trong serializer.

```
public class KryoOrder {

    private long orderId;
    private String item;
    private long orderTime;
    private String email;
    private Date updateTime;
    private KryoDetailOrder detailOrder;

    public static class KryoDetailOrder {
        private long amount;
        private String productCode;
    }

}
```

```
public class KryoOrderSerializer implements StreamSerializer<KryoOrder> {

    private static final ThreadLocal<Kryo> kryoThreadLocal = new ThreadLocal<Kryo>() {
        @Override
        protected Kryo initialValue() {
            Kryo kryo = new Kryo();
            kryo.register(Date.class);
            kryo.register(KryoOrder.class);
            kryo.register(KryoOrder.KryoDetailOrder.class);
            return kryo;
        }
    };
    @Override
    public int getTypeId() {
        return 1;
    }

    @Override
    public void write(ObjectDataOutput objectDataOutput, KryoOrder order) throws IOException {
        Kryo kryo = kryoThreadLocal.get();
        ByteArrayOutputStream byteArrayOutputStream =
                new ByteArrayOutputStream(16384);
        DeflaterOutputStream deflaterOutputStream =
                new DeflaterOutputStream(byteArrayOutputStream);
        Output output = new Output(deflaterOutputStream);
        kryo.writeObject(output, order);
        output.close();

        byte[] bytes = byteArrayOutputStream.toByteArray();
        objectDataOutput.write(bytes);
    }

    @Override
    public KryoOrder read(ObjectDataInput objectDataInput) throws IOException {
        InputStream in = new InflaterInputStream((InputStream) objectDataInput);
        Input input = new Input(in);
        Kryo kryo = kryoThreadLocal.get();
        return kryo.readObject(input, KryoOrder.class);
    }

    @Override
    public void destroy() {
    }
}
```

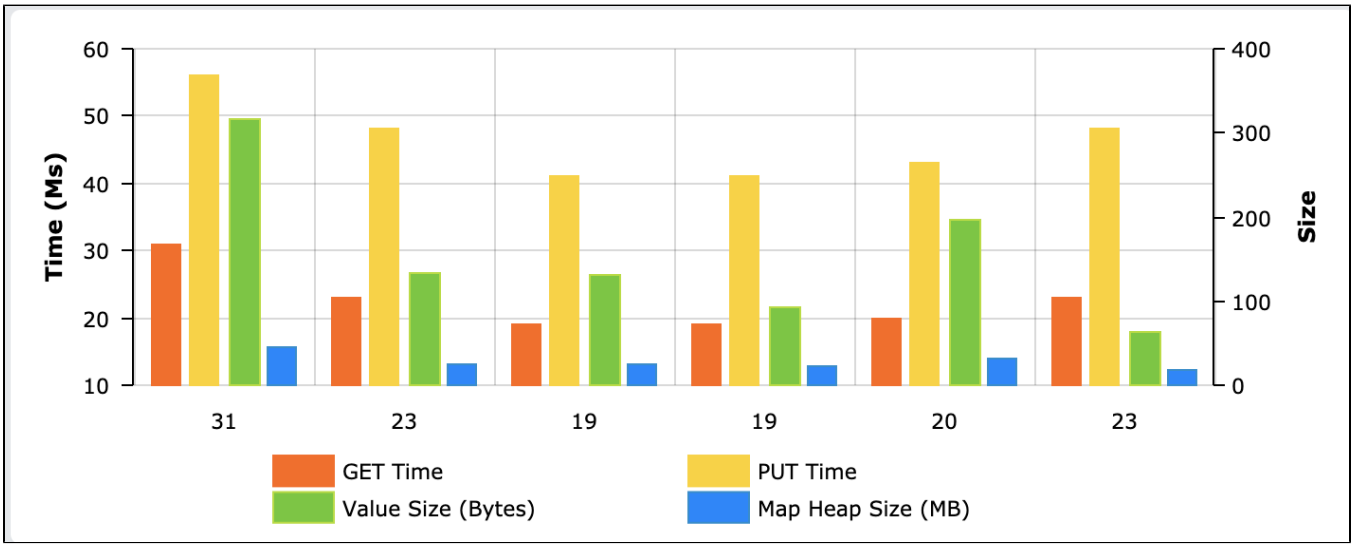
(Chi tiết cách cài đặt và cấu hình Serializer cho các object tham khảo tại: https://docs.hazelcast.org/docs/3.12.9/manual/html-single/index.html#serialization-configuration-wrap-up)

7.3.2. So sánh

- Thực hiện loadtest lần lượt các interface trên với thông số như sau:

- 2 node (2 server) trong cùng 1 mạng subnet.
- Mỗi server có cấu hình 1 Core CPU, 2GB RAM.
- Chạy service với option -Xmx1500M
- Test 2 step PUT & GET 100,000 phần tử vào Map

- Kết quả thu được:



Type	GET	PUT	ValueSize	MapHeapSize
Serializable	31	56	316	45
Externalizable	23	48	133	26
DataSerializable	19	41	131	26
IdentifiedSerializable	19	41	92	22
Portable	20	43	196	32
Kryo	23	48	64	19

(Hình 9. Thống kê performance khi thực hiện loadtest các interface serialize)

- Nhận xét

+ Về tổng quan, interface **IdentifiedDataSerializable** cho hiệu năng tốt nhất trong tất cả, mặc dù kích thước object được lưu có lớn hơn Kryo chút ít.

+ Interface **Serializable** "về chót bảng" trong tất cả mọi khía cạnh so sánh, tệ nhất về object size (gấp nhiều lần các interface khác). Lợi ích duy nhất là việc chỉ cần object implement interface là đã sử dụng được.

+ **Kryo** cho kết quả khá tốt về mọi khía cạnh, mặc dù tốc độ có thua 1 ít so với các interface của Hazelcast, tuy nhiên kích thước của object được lưu lại khá nhỏ. Lưu ý rằng Hazelcast lưu object in-memory, object càng nhỏ thì memory còn trống càng nhiều để có thể thực hiện các tác vụ khác, đồng thời khi sử dụng Kryo cũng không cần phải cài đặt chi tiết cách write/read của từng field trong object.

- Từ các đặc điểm trên, rút ra được bảng so sánh các interface sau đây:

Interface	Lợi ích	Hạn chế
Serializable	Interface cơ bản của Java, không cần cài đặt chi tiết	Tiêu thụ nhiều resource (CPU, memory), tốc độ chậm
Externalizable	Interface cơ bản của Java, hiệu năng tốt hơn **Serializable**	Cần cài đặt cách serialize các field
DataSerializable	Hiệu năng tốt hơn **Externalizable**	Cần cài đặt cách serialize các field; Sử dụng Reflection khi deserialize

IdentifiedDataSerializable	Hiệu năng tốt hơn **DataSerializable**; Không sử dụng Reflection khi deserialize	Cần cài đặt cách serialize các field; Cần cài đặt và cấu hình lớp Factory để khởi tạo object
Portable	Hiệu năng tốt hơn **Externalizable**; Không sử dụng Reflection khi deserialize; Hỗ trợ nhiều phiên bản của 1 clserializeVersionUIDass	Cần cài đặt cách serialize các field; Cần cài đặt và cấu hình lớp Factory để khởi tạo object
Custom Serializer	Không cần class object phải implement interface, tiện lợi khi sử dụng thư viện của bên thứ 3.	Cần cấu hình tương ứng cho class được serialize

8. Tổng kết

- Qua bài viết trên, các khái niệm cơ bản về Hazelcast cũng như cách sử dụng, áp dụng vào project đã được giới thiệu một cách tổng quan, ngoài ra có 1 số lưu ý trong quá trình sử dụng để đạt được hiệu năng (tốt nhất theo đánh giá của team) như sau:

- Tạo local cache cho các map Hazelcast để tăng performance khi truy xuất
- Dữ liệu khi cập nhật phải được serialize lại vào map (xem section 5.4)
- Khi shutdown service, cần gọi method hazelcastInstance.shutdown() để Hazelcast thực hiện Graceful Shutdown, nếu không có thể gặp exception khi các thread khác chưa shutdown hết vẫn cố truy cập vào dữ liệu trong Hazelcast cluster.
- Thêm field serializeVersionUID vào object model được serialize để đảm bảo object trước và sau serialize không bị khác version.

Chi tiết các tính năng mà bài viết đã tham khảo cũng như các tính năng nâng cao xem tại đây : https://docs.hazelcast.org/docs/3.12.9/manual/html-single/index.html