BÁO CÁO TUẦN 1

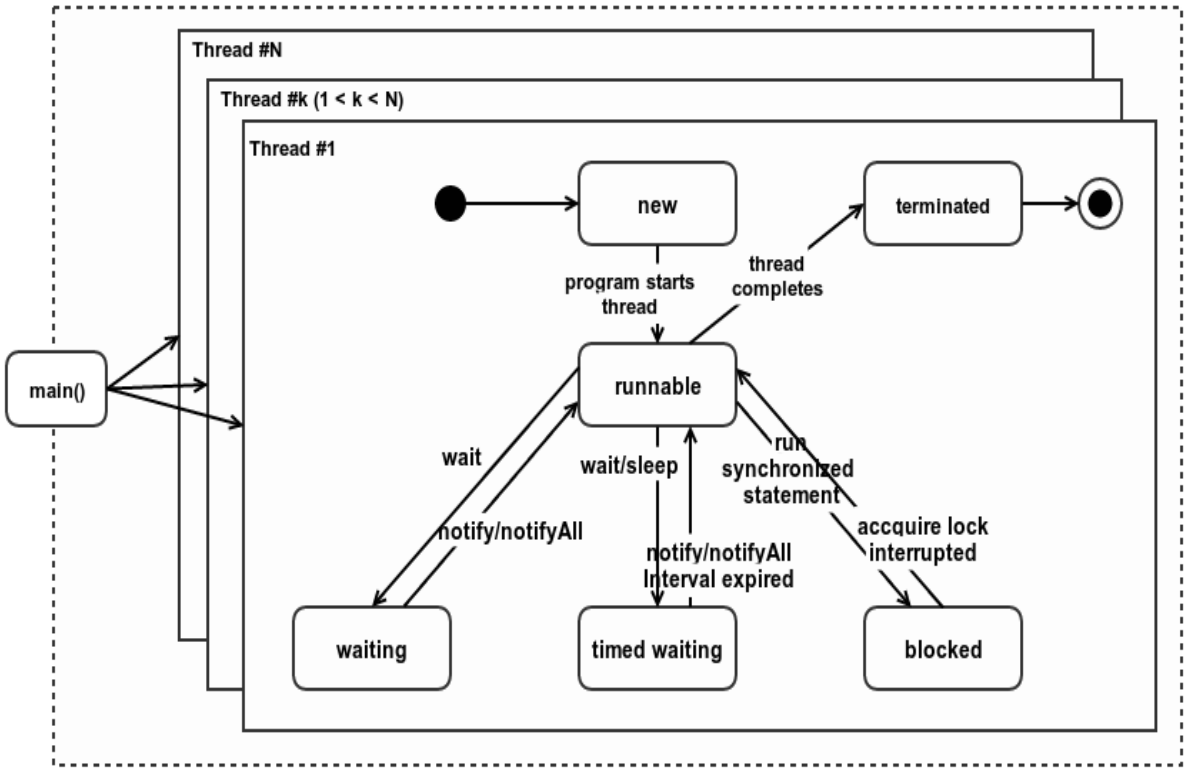
I. Java Core

1. Thread và Executor

Hầu hết các hệ điều hành ngày nay đều hỗ trợ lập trình đồng thời cho cả các *tiến trình* và *luồng*.

Tiến trình(Process) là quá trình hoạt động của ứng dụng độc lập với các ứng dụng khác. Trong một tiến trình có thể sử dụng nhiều luồng(thread) để thực thi code một cách đồng thời để tận dụng tối đa khả năng tất cả các nhân của CPU. Có thể coi thread là một process nhỏ chạy trong một chương trình và sử dụng chia sẻ các tài nguyên chung của chương trình đó. Điểm khác biệt của process và thread là: Các thread cùng một process thì dùng chung tài nguyên, dùng không gian bộ nhớ giống nhau, làm cho các thread có thể dễ dàng giao tiếp lẫn nhau, còn process thì không như vậy.

- *Vòng đời của một Thread:*



Một thread có các trạng thái khác nhau:new, runnable, waiting, timed waiting, blocked, terminated.

- New: Khi thread được tạo ra bằng toán tử new, trạng thái này tồn tại đến khi chương trình được kích hoạt.
- Runnable: Sau khi chương trình kích hoạt, thread chuyển sang trạng thái runnable, và thực thi các lệnh của nó.
- Waitting: Khi thread bị tạm ngừng thực thi và chờ trong khi một thread khác đang thực thi, thread này quay trở lại runnable khi nhận tín hiệu cho phép thực thi tiếp.
- Timed waitting: khi nó bị tạm thời ngừng thực thi và chờ trong một khaonrg thời gian định trước, nó quay lại runnable khi hết thời gian chờ hoặc nhận tín hiệu cho phép thực thi tiếp.
- Blocked: khi thread thực thi đến lệnh/khối lệnh đồng bộ mà lệnh/khối lệnh này đang được một thread khác thực thi.Nó quay trở lại runnable khi đến lượt nó được thực thi lệnh/khối lệnh đồng bộ.
- Terminated: khi thread thực hiện xong công việc. Chương trình sẽ loại bỏ những thread ở trạng thái này.

```
public class example {
    public static void main(String[] args) {
        // task cần thực thi
        Runnable task = () -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("The current thread: "+ threadName);
        };
        // thực thi thread main
        task.run();

        // tạo thread mới và thực thi thread
        Thread newThread = new Thread(task);
        newThread.start();

        System.out.println("Done");

    }
}
```

- *Runnable*:
  Trước khi start một thread cần xác định đoạn code sẽ được thực thi trong thread đó, gọi là *task*. Nó được thực thi bởi Runnable - một functional interface định nghĩa một hàm run() với kiểu trả về void, không có tham số.

Một kết quả tạo ra là:

```java
public class example {
    public static void main(String[] args) {
        Runnable runnable = () -> {
            try {
                String name = Thread.currentThread().getName();
                System.out.println("Foo " + name);

                // Tương đương với
                // Thread.sleep(5000);
                // Thread sẽ sleep trong 5s
                TimeUnit.SECONDS.sleep( timeout: 5);
                System.out.println("Bar " + name);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

Một kết quả khác xảy ra là "Done" sẽ được in ra cuối cùng. Chúng ta không thể đoán trước runnable sẽ in ra "Done" trước hay sau. Đó là do thực thi đồng thời các luồng với nhau.

- Thread có thể sleep trong một khoảng thời gian định trước qua TimeUnit hoặc Thread.sleep(time_sleep_millis):

```
The current thread: main
Done
The current thread: Thread-0

Process finished with exit code 0
```

Trong trường hợp trên, câu lệnh in thứ 2 sẽ bị chậm đi 5s so với câu lệnh in thứ nhất.

```java
import java.util.concurrent.*;

public class example {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() ->{
            String threadName = Thread.currentThread().getName();
            System.out.println("Hello " + threadName);
        });
    }
}
```

- *Excutors*:

```
Hello pool-1-thread-1
```

Thay vì làm việc trực tiếp với threads, chúng ta sử dụng Concurrency API *ExcutorService* giúp chạy các task không đồng bộ và quản lý pool các thread, vì vậy không cần phải tạo thread mới bằng tay. Chứng ta có thể chạy nhiều task suốt vòng đời của ứng dụng chỉ với một executor service.

Class Executors cung cấp các method thuận tiện cho việc tạo các loại executor service khác nhau. Sau đây là ví dụ sử dụng một executor với một thread pool với kích thước 1:

Ouput:

Khi chạy đoạn code trên thì java process sẽ không bao giờ dừng lại. ExecutorService cung cấp hai method:

- shutdown(): đợi task hiện tại thực thi xong rồi kết thúc.
- shutdownNow(): ngừng mọi task và tắt executor ngay lập tức.

Ví dụ:

```
attempt to shutdown executor
Hello pool-1-thread-1
shutdown finished

Process finished with exit code 0
```

```java
public class example {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("Hello " + threadName);
        });


        try {
            System.out.println("attempt to shutdown executor");
            executor.shutdown();
            executor.awaitTermination( timeout: 5, TimeUnit.SECONDS);
        }
        catch (InterruptedException e) {
            System.err.println("tasks interrupted");
        }
        finally {
            if (!executor.isTerminated()) {
                System.err.println("cancel non-finished tasks");
            }
            executor.shutdownNow();
            System.out.println("shutdown finished");
        }
    }
}
```

Kết qủa:

Executor chờ các task

đang chạy kết thúc rồi mới shutdown. Sau tối đa là 5s thì executor sẽ shutdown bằng cách interrupt tất cả các task đang chạy.

- *Callables và Futures:*

Ngoài Runnable thì executor còn hỗ trợ một loại task khác gọi là *Callable*. Callable là một functional interface giống như runnable, tuy nhiên thay vì kiểu void thì callable trả về một giá trị.

Các callable có thể gửi lên executor service giống như runnable. Tuy nhiên, executor service không thể trả về kết quả của callable trực tiếp. Thay vào đó, executor trả về một kết quả đặc biệt thuộc kiểu Future để có thể sử dụng khôi phục kết quả thật sau này.
Ví dụ:

```
Callable<Integer>  task = () -> {
    try  {
        TimeUnit.SECONDS.sleep( timeout: 1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};

ExecutorService executor = Executors.newFixedThreadPool( nThreads: 1);
// dùng method submit() tạo ra kết qảu kiểu Future
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());
try {
    Integer result = future.get();
    System.out.println("future done? " + future.isDone());
    System.out.print("result: " + result);
}
catch (InterruptedException e) {
    e.printStackTrace();
}
catch (ExecutionException e) {
    e.printStackTrace();
}
```

Kết qủa:

```
future done? false
future done? true
result: 123
```

Method get() sẽ block thread hiện tại và đợi đến khi callable hoàn thành trước khi trả về giá trị thực sự 123.
Lưu ý rằng các future chưa kết thúc sẽ sing ra ngoại lệ nếu chúng ta shutdown executor.

- *Timeouts:*
  future.get() sẽ block và đợi cho đến khi callbable kết thúc. Trong trường hợp

```java
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep( timeout: 2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

try {
    // Đặt timeout là 1s
    future.get( timeout: 1, TimeUnit.SECONDS);
}
catch (ExecutionException e) {
    e.printStackTrace();
}
catch (TimeoutException e) {
    e.printStackTrace();
}
catch (InterruptedException e) {
    e.printStackTrace();
}
```

xấu nhất, callable có thể chạy mãi mãi làm cho ứng dụng không phản hồi. Chúng ta có thể sử dụng timeout để xử lý:

Chúng ta đặt timeout là 1s. trong khi callable cần 2s trước khi thực hiện câu lệnh return. Vì vậy get() sẽ sinh ra TimeoutException:

```
java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
    at example.main(example.java:20)
```

- *InvokeAll*:

Excutor hỗ trợ submit nhiều callable một lúc bằng *invokeAll*(). Method này chấp nhận một collection callable và trả về một list các future. Ví dụ:

```
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<String>> callables = Arrays.asList(
        ()->"task1",
        ()->"task2",
        ()->"task3");

try {
    executor.invokeAll(callables) List<Future<String>>
            .stream() Stream<Future<String>>
            .map(future->{
                try {
                    return future.get();
                }
                catch (Exception e) {
                    throw new IllegalStateException(e);
                }
            }) Stream<String>
            .forEach(System.out::println);
}
catch (InterruptedException e) {
    e.printStackTrace();
}
```

- *InvokeAny*:

Không trả vệ một đối tượng future, invokeAny() block cho đến khi callable đầu tiên kết thúc và trả về kết quả của callable đó.

- *Scheduled Executor:*

Một ScheduledExecutorService có khả năng tạo schedule cho các task có thể chạy định kì hoặc một lần sau một khoảng thời gian nhất định.

Ví dụ: Lên lịch một task chạy sau 3s trôi qua:

```java
public static void main(String[] args) {
    ScheduledExecutorService excutor = Executors.newScheduledThreadPool( corePoolSize: 1);
    Runnable task = () -> System.out.println("scheduling:" + System.nanoTime());
    ScheduledFuture<?> future = excutor.schedule(task, delay: 3,TimeUnit.SECONDS);

    try {
        TimeUnit.MILLISECONDS.sleep( timeout: 1337);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }


    long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
    System.out.printf("Remaining Delay: %sms\n", remainingDelay);

}
```

Kiểu *ScheduledFuture* giống Future, tuy nhiên có thêm method getDelay() tả về thời gian delay còn lại. Khi chạy đoạn code trên ta thu được: Remaining Delay : 1661ms (gần bằng 3s - 1337ms = 1663ms). Tức là tại thời điểm getDelay(), vẫn còn 1661ms nữa thì task mới được thực thi.

Để lên lịch cho task chạy định kì thì dùng *scheduledAtFixedRate*() hoặc *scheduledWithFixedDelay*().

Hàm đầu tiên cho phép thực thi task với một tỷ lệ thời gian cố định. Ngoài ra còn cho phép chỉnh thời gian delay trước khi thực thi task lần đầu. Lưu ý rằng hàm này không tính đến thời gian thực tế của task. Vì vậy nếu chỉ định period là 1s nhưng task càn 2s để thực thi thì thread pool sẽ hoạt động sớm.

VD: *executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);*

Trong trường hợp này nên cân nhắc sử dụng hàm thứ 2 để thay thế. period của hàm này là khoảng thời gian từ khi kết thúc task này đến khi bắt đầu task tiếp theo. Tức là nếu period là 1s và task mất 2s để thực thi thì khoảng thời gian bắt đầu của 2 task liên tiếp là khoảng 3s.

VD: *executor.scheduledwithFixedDelay(task, initialDelay, period, TimeUnit.SECONDS);*

2. Độ phức tạp của các Java Collection:

Kí hiệu: h là dung lượng của Hash Table

- *List*:

  Là tập phần tử có thứ tự.

	Add	Remove	Get	Contains	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	Linked List
CopyonWriteArrayList	O(n)	O(n)	O(1)	O(n)	Array

- Set:

  Là tập gồm các phần tử không giống nhau.

	Add	Contains	Next	Data Structure
HashSet	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	Red-black Tree
CopyonWriteArraySet	O(n)	O(n)	O(1)	Array
ConcurrentSkipList	O(log n)	O(log n)	O(1)	Skip List

- Queue:

  Là tập các phần tử được thiết kế đẻ giữu các phần tử trước khi xử lý.

	Offer	Peak	Poll	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	Array
ArrayDequeue	O(1)	O(1)	O(1)	O(1)	Linked List
Concurrent-LinkedQueue	O(1)	O(1)	O(1)	O(n)	Linked List

ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)	Array
Prioririty Block-ingQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
SynchronousQueue	O(1)	O(1)	O(1)	O(1)	None!
DelayQueue	O(log n)	O(1)	O(log n)	O(1)	Priority Heap
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)	Linked List

- Map:

Là tập các cặp (key, value) trong đó các key là không giống nhau, và mỗi key chỉ có một value.

	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h/n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h/n)	Array
WeakHashMap	O(1)	O(1)	O(h/n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h/n)	Hash Tables
ConcurrentSkip-ListMap	O(log n)	O(log n)	O(1)	Skip List

3. Xử lý ngoại lệ:

3.1 Tổng quan:

Ngoại lệ là một lỗi có thể xảy ra trong qua trình thực thi chương trình và làm gián đoạn hoạt động bình thường của nó. Ngoại lệ có thể phát sinh từ dữ liệu sai, lỗi phần cứng, lỗi kết nối mạng,...

Khi thực thi chương trình, nếu gặp lỗi thì sẽ tạo ra đối tượng exception. Đối tượng ngoại lệ chứa nhiều thông tin debug. Trong một method nếu có ngoại lệ xảy ra thì sẽ ném ra một ngoại lệ cho môi trường runtime. Môi trường runtime sẽ tìm kiếm trong phương thức xảy ra lỗi, sau đó chuyển sang nơi gọi phương thức đó, để tìm ra khối lệnh xử lý ngoại lệ.

Java Exception chỉ xử lý ngoại lệ runtime chứ không xử lý lỗi khi compile.

3.2 Từ khóa:

+ **throw**: để ném ngoại lệ vào runtime để xử lý.

+ **throws**: Khi chúng ta ném một ngoại lệ trong phương thức mà không xử lý nó thì cần dùng *throws* trong chữ kí của phương thức để nơi gọi phương thức biết là ngoại lệ có thể ném ra từ phương thức đó. Phương thức gọi phương thức có ngoại lệ có thể xử lý hoặc ném ra ngoài cho một phương thức khác gọi nó. Có thể ném nhiều ngoại lệ bằng *throws* và có thể dùng cho phương thức main().

+ **try-catch**: dùng để xử lý ngoại lệ. Có thẻ có nhiều catch cho một try và có thể dùng các khối try-catch lồng nhau. Khối catch yêu cầu đối số nên thuộc kiểu Exception.

+ **finally**: (không bắt buộc) chỉ có thể sử dụng với khối try-catch. Khối finally luôn luôn được thực thi dù cho ngoại lệ có xảy ra hay không.

Lưu ý:

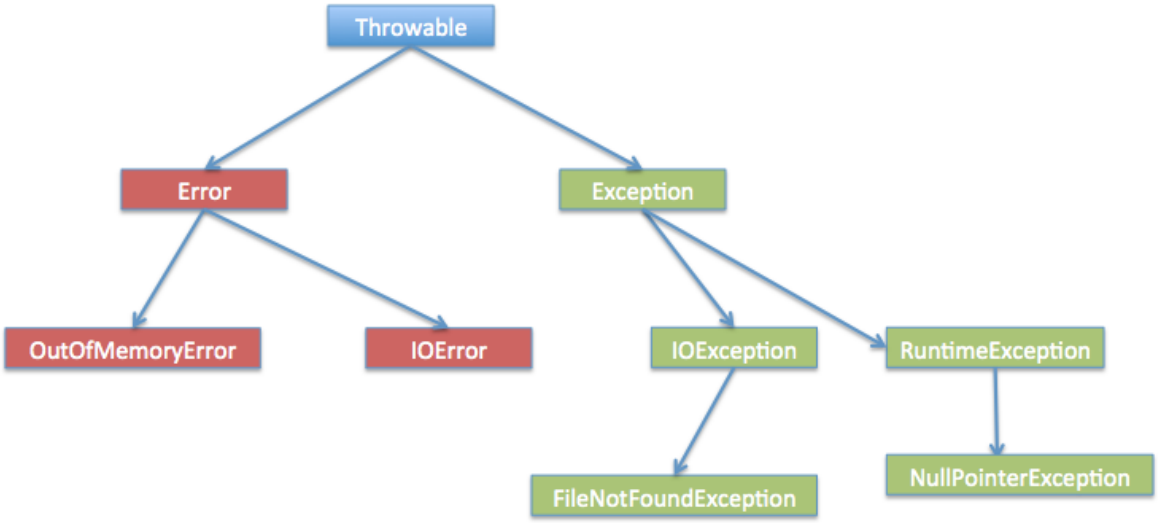
- không thể dùng catch và finally nếu không có try.
- try nên có catch hoặc finally, có thể dùng cả hai được.
- không thể viết bất cứ dòng code nào giữa try-catch-finally.
- 1 try có thể có nhiều catch.
- try-catch có thẻ lồng nhau.
- mỗi câu lệnh try-catch chỉ có duy nhất một finally.

3.3 Phân cấp:

Throwable là lớp cha của phân cấp Java Exception, nó có hai lớp con là Error và Exception. Exception còn được chia ra ngoại lệ kiểm tra được và ngoại lệ runtime.

- **Error**: là các tình huống đặc biệt nằm ngoài phạm vi của ứng dụng và không thể dự đoán và phục hồi từ chúng. Ví dụ lỗi phần cứng, lỗi JVM, lỗi tràn bộ nhớ,... Đó là lý do tại sao chúng ta có một hệ thống phân cấp lỗi riêng biệt và chúng ta không nên cố gắng xử lý các tình huống này. Một số lỗi phổ biến là OutOfMemoryError và StackOverflowError.

- **Checked Exception:** là các tình huống đặc biệt mà chúng ta có thể dự đoán trong một chương trình và cố gắng khôi phục từ đó, ví dụ FileNotFoudException. Chúng ta nên cử lý ngoại lệ này và cung cấp thông báo hữu ích cho người dùng. Exception là lớp cha của tất cả Checked Exception và nếu chúng ta throw 1 Checked Exception thì chúng ta phải xử lý trong phương thức đó hoặc ném ngoại lệ ra ngoài bằng từ khóa throws.
- **Runtime Exception**: do một thao tác lập trình kém, ví dụ như cố truy xuất một phần từ mảng. RuntimeException là lớp cha của mọi runtime exception.



Nếu chúng ta ném một runtime exception trong một phương thức, thì không bắt buộc phải dùng throws trong chữ kí của phương thức đó. Runtime Exception có thể tránh được nhờ lập trình tốt hơn.

3.4 Các phương thức hữu ích:

Do lớp Throwable định nghĩa.

- **public String getMessage()**
- **public String getLocalizedMessage()**
- **public synchronized Throwable getCause()**
- **public String toString()**
- **public void printStackTrace()**

3.5 Quản lý khối tài nguyên tự động và cải tiến khối Catch trên Java 7

Java 7 cung cấp tính năng có thể catch nhiều EXception trong một khối try.

Ví dụ:

```
catch(IOException | SQLException ex){
    logger.error(ex);
    throw new MyException(ex.getMessage());
}
```

Dùng try-with-resource để quản lý tài nguyên:

```
try (MyResource mr = new MyResource()) {
        System.out.println("MyResource created in try-with-resources");
    } catch (Exception e) {
        e.printStackTrace();
    }
```

Sau khi thoát khỏi khối try-catch thì tài nguyên sẽ được tự động đóng.

3.6 Các lưu ý:

- Dùng các Exception cụ thể.
- Ném ngoại lệ sớm nếu có thể.
- Bắt ngoại lệ muộn: Chỉ nên bắt ngoại lệ khi chúng ta có thể xử lý nó một cách thích hợp.
- Đóng tài nguyên.
- Ghi nhật kí ngoại lệ.
- Dùng một khối catch để bắt nhiều ngoại lệ.
- Dùng Ngoại lệ tùy chỉnh.
- Quy ước đặt tên với ngoại lệ tùy chỉnh : Ngoại lệ tùy chỉnh kết thúc bằng 'Exception'
- Dùng ngoại lệ thận trọng: Đôi khi không cần phải ném ngoại lệ vì không cần thiết, thay vào đó có thể trả về giá trị boolean để giải quyết.
- Document ngoại lệ đã ném: Dùng javadoc @throw để xác định rõ các ngoại lệ đã ném ra bởi phương thức.

II. N-gram và tf-idf

1. N-gram

- *Khái niệm:*

Là mô hình ngôn ngữ thống kê cho phép ước lượng xác suất cho một chuỗi m phần tử (thường là từ) P(w1w2...wm) tức là cho phép dự đoán khả năng một chuỗi từ xuất hiện trong ngôn ngữ đó.

Theo công thức Bayes:

P(AB) = P(A|B) * P(A)

Suy ra:

P(w1w2…wm) = P(w1) * P(w2|w1) * P(w3|w1w2) *…* P(wm|w1w2…wm-1)

Theo công thức này thì bài toán tính xác suất của một chuỗi từ với điều kiện biết các từ trước nó.

Trong thực tế , dựa vào giả thuyết Markov người ta chỉ tính xác suất của một từ dựa vào nhiều nhất n từ xuất hiện liền trước nó, thông thường n=0,1,2,3. Vì vậy người ta gọi là mô hình n-gram, trong đó n là số lượng từ bao gồm cả từ cần tính và các ngữu cảnh phía trước.

Nếu áp dụng xấp xiwr Markov, xác suất xuát hiện của một từ wm được coi như chỉ phụ thuộc vào n từ đứng liền trước nó (wm-n...wm-1). Suy ra:

P(w1w2…wm) = P(w1) * P(w2|w1)*… * P(wm-1|wm-n-1 wm-n …wm-2)
*P(wm|wm-n wm-n+1 …wm-1)

- *Phân loại:*

+ n=1: unigram.

+ n=2: digram.(được sử dụng nhiều trong phân tích hình thái từ).

+ n=3: trigram. Do n càng lớn thì số trường hợp vàng nhiều nên người ta thường dùng n=1, 2 đôi khi là 3.

- Công thức tính xác suất thô:

gọi $C(w_{i-n+1}...w_{i-1}w_i)$ là tần số xuất hiện của cụm $w_{i-n+1}...w_{i-1}w_i$ trong tập văn bản huấn luyện.

Gọi $P(w_i|w_{i-n+1}...w_{i-1})$ là xác suất w_1 đi sau cụm $w_{i-n+1}...w_{i-1}$

Ta có:

$P(w_i|w_{i-n+1}...w_{i-1}) = C(w_{i-n+1}...w_{i-1}w_i) / C(w_{i-n+1}...w_{i-1})$

- *Các vấn đề:*

- Phân bố không đều. => làm mịn.

- Kích thước bộ nhớ của mô hình ngôn ngữ.

- Các phương pháp làm mịn:

Do sự phân bố không đều trong tập văn bản huấn luyện (Ví dụ 1 cụm n-

gram chưa xuất hiện trong tập huấn luyện bao giờ dẫn đến xác suất bằng 0)
Phân loại:

- Chiết khấu: giảm lượng nhỏ xác suất của các cụm n-gram có xác suất lớn hơn 0 để bù cho các cụm n-gram không xuất hiện trong tập huấn luyện.
- Try hồi: tính toán xác suất các cụm gram không xuất hiện trong tập huấn luyện dựa vào các cum n-gram thành phần có đọ đài ngắn hơn và có xác suất lớn hơn 0.
- Nội suy: tính toán xác suất cảu tất cả các cụm n-gram dựa vào xác suất của cụm n-gram ngắn hơn.

- *Thuật toán làm mịn Add-One:*
  Thuộc nhóm chiết khấu
  - Phương pháp này sẽ cộng thêm vào số lần xuất hiện của mỗi cụm n-gram lên 1, khi đó xác suất của cụm n-gram sẽ được tính lại là:

    p = (c+1) / (n+v)

  Trong đó:
  - c là số lần xuất hiện cụm n-gram trong tập ngữ liệu mẫu,
  - n là số cụm N-gram,
  - v là kích thước của toàn bộ từ vựng

  - Để hạn chế giảm xác suất của các cụm N-gram quan trọng chúng ta bổ sung thêm hê số alpha

    p = (c+alpha) / (n + alpha*v)

2. Tf-idf:

là viết tắt của term frequency - inverse document frequency.

$$\mathrm{tf}(t,d) = \frac{\mathrm{f}(t,d)}{\max\{\mathrm{f}(w,d) : w \in d\}}$$

+ **tf-idf** của một từ là một con số thu được qua thống kê này trong thể hiện mức độ quan trọng của từ một văn bản, mà bản thân văn bản đang xét nằm trong một tập văn bản.

+**tf** là tần số xuất hiện một từ trong văn bản:

- Thương của số lần xuất hiện một từ trong văn bản và số lần xuất hiện nhiều nhất của một từ bất kì trong văn bản.
- **f(t,d)** - số lần xuất hiện từ t trong văn bản **d**.
- **max{f(w,d):w€d}** - số lần xuất hiện nhiều nhất của một từ bất kỳ

trong văn bản.

+ **IDF** – *inverse document frequency.* Tần số nghịch của 1 từ trong tập văn bản (corpus).

Tính **IDF** để giảm giá trị của những từ phổ biến. Mỗi từ chỉ có 1 giá trị **IDF** duy nhất trong tập văn bản.

- |D| - tổng số văn bản trong tập **D**
- |{d € D : t € d }|:- số văn bản chứa từ nhất định, với điều kiện xuất hiện trong văn bản d.
- Nếu từ đó không xuất hiện ở bất cứ 1 văn bản nào trong tập thì mẫu số sẽ bằng 0 => phép chia cho không không hợp lệ, vì thế người ta thường thay bằng mẫu thức |{d € D : t € d}|

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D)$$

Cơ số logarit trong công thức này không thay đổi giá trị của 1 từ mà chỉ thu hẹp khoảng giá trị của từ đó. Vì thay đổi cơ số sẽ dẫn đến việc giá trị của các từ thay đổi bởi một số nhất định và tỷ lệ giữa các trọng lượng với nhau sẽ không thay đổi. (nói cách khác, thay đổi cơ số sẽ không ảnh hưởng đến tỷ lệ giữa các giá trị IDF). Tuy nhiên việc thay đổi khoảng giá trị sẽ giúp tỷ lệ giữa IDF và TF tương đồng để dùng cho công thức TF-IDF như bên dưới.

Giá trị **TF-IDF**:

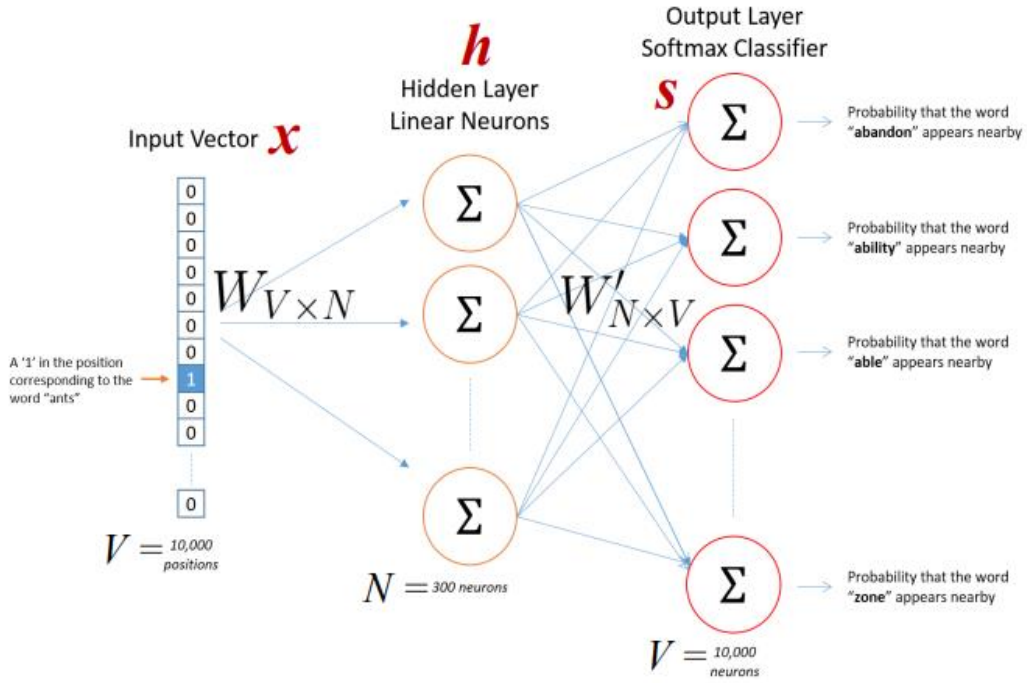
Những từ có giá trị TF-IDF cao là những từ xuất hiện nhiều trong văn bản này, và xuất hiện ít trong các văn bản khác. Việc này giúp lọc ra những từ phổ biến và giữ lại những từ có giá trị cao (từ khoá của văn bản đó).

$$\mathrm{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

III. Word2vec

Trong natural language processing thì để xử lý dữ liệu text chúng ta cũng phải chuyển dữ liệu từ text sang numeric, tức là đưa nó vào một không gian mới người ta thường gọi là embbding. Trước đây người ta mã hóa theo kiểu one hot encoding tức là tạo một vocabualary cho dữ liệu và mã hóa các word trong document thành những vectoc, nếu word đó có trong document thì mã hóa là 1 còn không có sẽ là 0. Kết quả tạo ra một sparse matrix, tức là matrix hầu hết là 0.Các mã hóa này có nhiều nhược điểm đó là thứ nhất là số chiều của nó rất lớn (NxM, N là số document còn M là số vocabulary), thứ 2 các word không có quan hệ với nhau.

- Word2vec được tạo ra năm 2013 bởi một kỹ sư ở google có tên là **Tomas Mikolov**. Nó là một model unsupervised learning,được training từ large corpus . Chiều của Word2vec nhỏ hơn nhiều so với one-hot-encoding, với số chiều là NxD với N là Number of document và D là số chiều word embedding . Word2vec có 2 model là skip-gram và Cbow.
    - Skip-gram model là model predict word surrounding khi cho một từ cho trước, ví dụ như text = "I love you so much". Khi dùng 1 window search có size 3 ta thu được : {(i,you),love},{(love,so),you},{(you,much),so}. Nhiệm vụ của nó là khi cho 1 từ center ví dụ là love thì phải predict các từ xung quang là i, you.
    - Cbow là viết tắt của continous bag of word . Model này ngược với model skip-gram tức là cho những từ surrounding predict word current.
    - Trong thực tế người ta chỉ chọn một trong 2 model để training, Cbow thì training nhanh hơn nhưng độ chính xác không cao bằng skip-gram và ngược lại
- Skip-gram:



- Input là one-hot-vector mỗi word sẽ có dạng x1,x2,...xv trong đó V là số vocab-ulary, là một vector trong đó mỗi word sẽ có giá trị 1 tương đương với index trong vocabulary và còn lại sẽ là 0.
- Weight matrix giữa input và hidden layer là matrix W(có dimention VxN) có ac-tive function là linear, weight giữa hidden và out put là W′ (có dimention là NxV) active function của out put là soft max.
- Mỗi row của W là vector N chiều đại diện cho v_ω là mỗi word trong input layer.Mỗi row của W là $v_\omega T$. Lưu ý là input là 1 one hot vector (sẽ có dạng 000100) chỉ có 1 phần tử bằng 1 nên.

$$h=W^Tx=v_\omega{}^T$$

- Từ hidden layer đến out put là matrix $W'' = \omega'_{i,j}$. Ta tính score u_i cho mỗi word trong vocabulary.

$$u_j=v'_{\omega j}h$$

- Trong đó $v_{\omega j}$ là vector colum j trong W′. Tiếp đó ta sử dụng soft max funtion.
- Ouput là một vector duy nhất (V thành phần, chứa - Word2vec sẽ train một mạng neural đơn giản với chỉ một hidden layer để thực hiện một nhiệm vụ nhất định, nhưng sau đó chúng ta không thực sự sử dụng mạng neural đó cho task đã train. Mục tiêu của chúng ta chỉ là tìm hiểu các trọng số trong hidden layer.

- Tham số "Window size" : số từ phía sau / phía trước của từ đó
+ Ví dụ : window size = 5 : xét phạm vi 5 từ liên tiếp phía trước đến 5 từ liên tiếp phía sau từ đang xét

- Giữa Input Layer và Hidden Layer được biểu diễn bằng ma trận trọng số W (V

* N)
+ V rows (một hàng cho một từ trong tập từ vựng) - giả từ tập từ vựng có V từ
+ N columns (mỗi cột cho một hidden neural) - N là số features
+ Khi đó mỗi hàng trong ma trận trọng số thực sự là vector từ chúng ta cần => Tìm được ma trận trọng số này

- Ưu điểm :
+ Hidden layer hoạt động như một bảng tra cứu: chúng ta chỉ cần hàng của ma trận ứng với 1

=> Đầu ra của hidden layer là word vector cho từ đầu vào

- Output layer :
+ Là một softmax regression classifier
+ Ouput của hidden layer là word vector 1*300
+ Ouput của ouput layer là xác suất của 1 từ xuất hiện gần từ đầu vào

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$