Studio sull'uso di Java per la programmazione di una blockchain

Luca Vicentini - VR408207

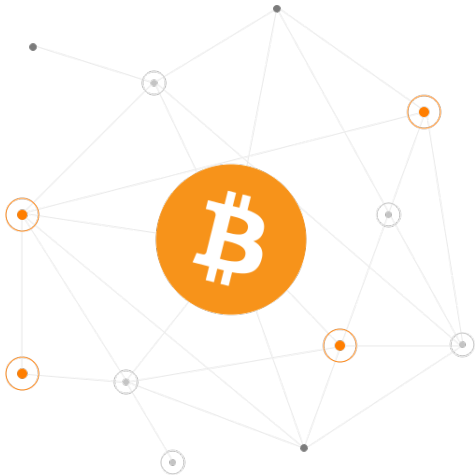
21 marzo 2019

Relatore: Nicola Fausto Spoto

Indice

Bitcoin (1/2)

- Tecnologia **peer-to-peer** che non fa uso di un'autorità centrale.
- L'emissione di nuova valuta e la gestione delle transazioni viene effettuata collettivamente dalle rete.



Ogni utente crea un **indirizzo BTC** che utilizzerà per effettuare/ricevere pagamenti che vengono registrati in **transazioni**.

Bitcoin (2/2)

Miner e processo di Mining

- **Conferma le transazioni** includendole in un blocco.
- Tale blocco rispetta delle regole crittografiche molto rigide che impediscono che qualunque blocco precedente venga modificato.
- Il blocco verrà aggiunto alla **blockchain**.

Proof-of-Work è una sfida globale che comporta l'effettuare ripetutamente l'hash dell'header del blocco, variando un numero casuale, fino a trovare una soluzione con un certo pattern.

Ethereum (1/2)

- Infrastruttura open-source, globalmente decentralizzata, per eseguire **Smart Contract**.
- Utilizza la blockchain per sincronizzare e memorizzare le modifiche di stato del sistema e la cripto-valuta **ether**.

Gli smart contract vengono eseguiti all'interno della Ethereum Virtual Machine (**EVM**) e presentano un **linguaggio turing-completo**.

Ethereum (2/2)

La Turing-completezza del linguaggio obbliga l'EVM a **limitare l'esecuzione dei contratti** implementando un meccanismo di misurazione delle risorse chiamato **gas**.

Solidity

Linguaggio di programmazione **contract-oriented** che compila dei programmi scritti in Solidity nel linguaggio bytecode per l'EVM.

Questo linguaggio è tuttora in via di perfezionamento e soffre la mancanza di strumenti d'aiuto al programmatore.

Takamaka: A Java Framework for Smart Contracts

Framework il cui scopo è **semplificare lo sviluppo** di smart contract sfruttando competenze e strumenti già esistenti del mondo *Java*.

- Takamaka utilizza un sottoinsieme delle librerie *Java*, chiamate **white-listed**.
- Le analisi effettuate mirano a trovare quei metodi che presentano comportamenti non deterministici.

L'esito di uno smart contract dovrà risultare sempre lo stesso dato il contesto della transazione e lo stato della blockchain in quel momento.

Analisi della Classe java.lang.Object

- **hashCode()** Restituisce l'hash code dell'oggetto.
  Calcolato utilizzando l'indirizzo in memoria di tale oggetto.

```
1  public int foo() {
2     return new Object().hashCode();
3  }
```

- **toString()** Stringa che rappresenta testualmente l'oggetto.
  `getClass().getName() +'@'+ Integer.toHexString(hashCode())`

```
1  public String foo() {
2     return new Object().toString();
3  }
```

L'esecuzione dei metodi `foo()` risulta non deterministica

Analisi della Classe java.util.HashSet e java.util.HashMap

Non forniscono garanzie sull'ordine di iterazione del `Set`/`Map`.
Non è garantito che tale ordine rimanga costante nel tempo.

- **iterator()** Restituisce in iteratore sugli elementi del `Set`, gli elementi non vengono restituiti con un ordine particolare.
- **values()** Ritorna una `Collection view` di tutti i valori contenuti all'interno della `Map`.

```
1   private HashSet h1 = new HashSet();
2   private Map<String, String> h2 = new HashMap<>();
3   public Iterator getIterator1() {
4      return h1.iterator();
5   }
6   public Iterator getIterator2() {
7      return h2.values().iterator();
8   }
```

Analisi della Classe java.util.Stream (1/3)

Una `Stream` è una sequenza di elementi che supporta operazioni di aggregazione sequenziali e parallele. Introdotte da `Java 8`.

- **findAny()** Restituisce un qualche elemento all'interno della `Stream`. Operazione esplicitamente non deterministica.

```
1  public String foo() {
2     List<String> lst = Arrays.asList( ... );
3
4     return lst.stream().filter(s -> s.startsWith("J"))
5        .findAny().get();
6  }
```

Utilizzare `findFirst()` per avere un risultato deterministico.

Analisi della Classe java.util.Stream (2/3)

- **forEach(...)** esegue un'azione per ciascun elemento della `Stream`. Il comportamento esplicitamente non deterministico.

```
1  public void foo() {
2     List<String> lst = Arrays.asList( ... );
3
4     lst.stream().filter(s -> s.startsWith("J"))
5        .forEach(e -> System.out.print(e + " "));
6  }
```

Non siamo riusciti a trovare un esempio di funzionamento anomalo che provi il non determinismo, ma non può voler dire che tale metodo ritorni sempre i risultati aspettati, bensì che **non possiamo contare su di esso**.

Analisi della Classe java.util.Stream (3/3)

- **forEachOrdered(...)** esegue un'azione per ciascun elemento della `Stream` in base all'ordine di incontro, se la `Stream` ne ha definito uno.

```
1  public void foo() {
2     List<String> lst = Arrays.asList( ... );
3
4     lst.stream().filter(s -> s.startsWith("J"))
5        .forEachOrdered(e -> System.out.print(e + " "));
6  }
```

- La stream deve avere un'ordine definito.
- Si devono utilizzare solo collezioni ordinate.

Analisi della Classe java.lang.System (1/2)

- **currentTimeMillis()** ritorna un `long`, rappresenta il tempo corrente misurato in millisecondi trascorsi dal 00:00:00 01/01/1970 UTC.

```
1   public long getCurrentMillis() {
2      return System.currentTimeMillis();
3   }
```

- Metodo non deterministico
- Utilizzato da `java.util.Date`, `java.time.Clock` e `java.util.Time`.

Analisi della Classe java.lang.System (2/2)

- **nanoTime()** ritorna un valore temporale di esecuzione della JVM misurata in nanosecondi.

```
1 |   public long getNanoTime() {
2 |      return System.nanoTime();
3 |   }
```

- Metodo non deterministico
- Utilizzato per esempio da `java.util.Random`.

Analisi della Classe java.util.Random (1/2)

Genera una `Stream` di numeri pseudocasuali a partire da un seme `seed` modificato ad ogni invocazione.

- **new Random()** setta un valore di `seed` distinto per qualsiasi chiamata, diverso ad ogni invocazione.

```
1   public int rnd() {
2      return new Random().nextInt();
3   }
```

Il valore di `seed` è derivato da `System.nanoTime()` e **sarà diverso per ogni JVM**.

Analisi della Classe java.util.Random (2/2)

- **new Random(int seed)** setta il valore di `seed` indicato che influenzerà i valori generati.

```
1   public int rnd() {
2      return new Random(1993).nextInt();
3   }
```

58 83 97 81 90 70 41 94 80 79 51 83 32 28 73 ...

- Comportamento deterministico
- Restituisce sempre la stessa sequenza

In sintesi

java.lang.Object	hashCode()	✗
	toString()	✗
java.util.HashSet	iterator()	✗
java.util.HashMap	values().iterator()	✗
java.util.Stream	findAny()	✗
	findFirst()*	✓
	forEach(...)	✗
	forEachOrdered(...)*	✓
	peek()	✗
java.lang.System	currentTimeMillis()	✗
	nanoTime()	✗
java.util.Random	Random()	✗
	Random(int seed)	✓

✗ metodi per la concorrenza e `parallelStream()`.

Fine
Grazie per l'attenzione