Previous Next

535. Encode and Decode TinyURL

March 18, 2017 | 53.4K views

Average Rating: 4.90 (49 votes)

> Note: This is a companion problem to the System Design problem: Design TinyURL.

TinyURL is a URL shortening service where you enter a URL such as `https://leetcode.com/problems/design-tinyurl` and it returns a short URL such as `http://tinyurl.com/4e9iAk`.

Design the `encode` and `decode` methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.

Solution

Approach #1 Using Simple Counter[Accepted]

In order to encode the URL, we make use of a counter(i), which is incremented for every new URL encountered. We put the URL along with its encoded count(i) in a HashMap. This way we can retrieve it later at the time of decoding easily.

Java

```java
public class Codec {
    Map<Integer, String> map = new HashMap<>();
    int i = 0;

    public String encode(String longUrl) {
        map.put(i, longUrl);
        return "http://tinyurl.com/" + i++;
    }

    public String decode(String shortUrl) {
        return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));
    }
}
```

Performance Analysis

- The range of URLs that can be decoded is limited by the range of int.
- If excessively large number of URLs have to be encoded, after the range of int is exceeded, integer overflow could lead to overwriting the previous URLs' encodings, leading to the performance degradation.
- The length of the URL isn't necessarily shorter than the incoming longURL. It is only dependent on the relative order in which the URLs are encoded.
- One problem with this method is that it is very easy to predict the next code generated, since the pattern can be detected by generating a few encoded URLs.

Approach #2 Variable-length Encoding[Accepted]

Algorithm

In this case, we make use of variable length encoding to encode the given URLs. For every longURL, we choose a variable codelength for the input URL, which can be any length between 0 and 61. Further, instead of using only numbers as the Base System for encoding the URLSs, we make use of a set of integers and alphabets to be used for encoding.

Java

```java
public class Codec {

    String chars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    HashMap<String, String> map = new HashMap<>();
    int count = 1;

    public String getString() {
        int c = count;
        StringBuilder sb = new StringBuilder();
        while (c > 0) {
            c--;
            sb.append(chars.charAt(c % 62));
            c /= 62;
        }
        return sb.toString();
    }

    public String encode(String longUrl) {
        String key = getString();
        map.put(key, longUrl);
        return "http://tinyurl.com/" + key;
        count++;
    }

    public String decode(String shortUrl) {
        return map.get(shortUrl.replace("http://tinyurl.com/", ""));
    }
}
```

Performance Analysis

- The number of URLs that can be encoded is, again, dependent on the range of int, since, the same $count$ will be generated after overflow of integers.
- The length of the encoded URLs isn't necessarily short, but is to some extent dependent on the order in which the incoming longURL's are encountered. For example, the codes generated will have the lengths in the following order: 1(62 times), 2(62 times) and so on.
- The performance is quite good, since the same code will be repeated only after the integer overflow limit, which is quite large.
- In this case also, the next code generated could be predicted by the use of some calculations.

Approach #3 Using hashcode[Accepted]

Algorithm

In this method, we make use of an inbuilt function $\text{hashCode}()$ to determine a code for mapping every URL. Again, the mapping is stored in a HashMap for decoding.

The hash code for a String object is computed(using int arithmetic) as −

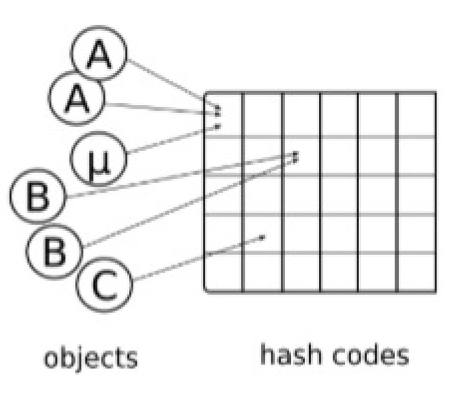
$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$, where s[i] is the ith character of the string, n is the length of the string.

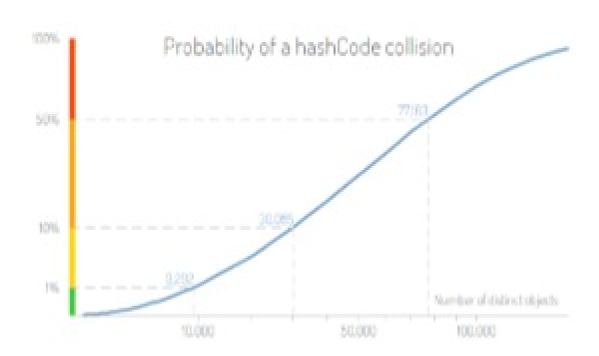
Java

```java

public class Codec {
    Map<Integer, String> map = new HashMap<>();

    public String encode(String longUrl) {
        map.put(longUrl.hashCode(), longUrl);
        return "http://tinyurl.com/" + longUrl.hashCode();
    }

    public String decode(String shortUrl) {
        return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));
    }
}

```

Performance Analysis

- The number of URLs that can be encoded is limited by the range of int, since hashCode uses integer calculations.
- The average length of the encoded URL isn't directly related to the incoming longURL length.
- The $\text{hashCode}()$ doesn't generate unique codes for different string. This property of getting the same code for two different inputs is called collision. Thus, as the number of encoded URLs increases, the probability of collisions increases, which leads to failure.
- The following figure demonstrates the mapping of different objects to the same hashcode and the increasing probability of collisions with increasing number of objects.





- Thus, it isn't necessary that the collisions start occuring only after a certain number of strings have been encoded, but they could occur way before the limit is even near to the int. This is similar to birthday paradox i.e. the probability of two people having the same birthday is nearly 50% if we consider only 23 people and 99.9% with just 70 people.
- Predicting the encoded URL isn't easy in this scheme.

Approach #4 Using random number[Accepted]

Algorithm

In this case, we generate a random integer to be used as the code. In case the generated code happens to be already mapped to some previous longURL, we generate a new random integer to be used as the code. The data is again stored in a HashMap to help in the decoding process.

Java

```java

public class Codec {
    Map<Integer, String> map = new HashMap<>();
    Random r = new Random();
    int key = r.nextInt(Integer.MAX_VALUE);

    public String encode(String longUrl) {
        while (map.containsKey(key)) {
            key = r.nextInt(Integer.MAX_VALUE);
        }
        map.put(key, longUrl);
        return "http://tinyurl.com/" + key;
    }

    public String decode(String shortUrl) {
        return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));
    }
}

```

Performance Analysis

- The number of URLs that can be encoded is limited by the range of int.
- The average length of the codes generated is independent of the longURL's length, since a random integer is used.
- The length of the URL isn't necessarily shorter than the incoming longURL. It is only dependent on the relative order in which the URLs are encoded.
- Since a random number is used for coding, again, as in the previous case, the number of collisions could increase with the increasing number of input strings, leading to performance degradation.
- Determining the encoded URL isn't possible in this scheme, since we make use of random numbers.

Approach #5 Random fixed-length encoding[Accepted]

Algorithm

In this case, again, we make use of the set of numbers and alphabets to generate the coding for the given URLs, similar to Approach 2. But in this case, the length of the code is fixed to 6 only. Further, random characters from the string to form the characters of the code. In case, the code generated collides with some previously generated code, we form a new random code.

Java

```java

public class Codec {
    String alphabet = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    HashMap<String, String> map = new HashMap<>();
    Random rand = new Random();
    String key = getRand();

    public String getRand() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 6; i++) {
            sb.append(alphabet.charAt(rand.nextInt(62)));
        }
        return sb.toString();
    }

    public String encode(String longUrl) {
        while (map.containsKey(key)) {
            key = getRand();
        }
        map.put(key, longUrl);
        return "http://tinyurl.com/" + key;
    }

    public String decode(String shortUrl) {
        return map.get(shortUrl.replace("http://tinyurl.com/", ""));
    }
}

```

Performance Analysis

- The number of URLs that can be encoded is quite large in this case, nearly of the order $(10 + 26 * 2)^6$.
- The length of the encoded URLs is fixed to 6 units, which is a significant reduction for very large URLs.
- The performance of this scheme is quite good, due to a very less probability of repeated same codes generated.
- We can increase the number of encodings possible as well, by increasing the length of the encoded strings. Thus, there exists a tradeoff between the length of the code and the number of encodings possible.
- Predicting the encoding isn't possible in this scheme since random numbers are used.

Rate this article:

Previous Next

Comments: 15 Sort By

Type comment here... (Markdown is supported)

Preview Post

thepatelguy ★ 51 January 23, 2019 10:46 AM

Approach 2, how come there is `count++` after the `return` statement in encode() and it is an accepted answer?

30 Share Reply

juanqiuxiaoxiao ★ 14 June 25, 2018 1:33 AM

In Method 1, why don't we start from i = Integer.MIN_VALUE?

13 Share Reply

SHOW 5 REPLIES

runningsnail ★ 112 August 27, 2017 7:33 AM Report

Do we need to de-dupe the long url? Can one same long url have multiple tinyUrls?

6 Share Reply

ghudeihed ★ 4 August 5, 2018 4:16 AM

I used a Guid for the key and answer was accepted

```
public class Codec {
    private Dictionary<string,string> hashTable = new Dictionary<string,string>
();
```

Read More

4 Share Reply

fuyaoli ★ 95 April 19, 2018 9:38 AM

the Approach#2 doesn't work actually, what's wrong with it ?

3 Share Reply

SHOW 2 REPLIES

ricace ★ 51 April 8, 2019 4:49 AM

really nice solutions. Thanks the author

2 Share Reply

stella_ ★ 11 February 26, 2018 5:50 AM

for Approach #2 Variable-length Encoding, it s not 1:1 mapping. there will have duplicate converted tiny url since every 62 there '!' == '_' (33+62=95). should there also need a hashmap storing these duplicates ?

1 Share Reply

SHOW 1 REPLY

Govind93 ★ 65 September 11, 2017 12:58 AM

Great Step by Step Solution

1 Share Reply

Novarg ★ 4 October 29, 2019 8:50 PM

How on earth is this not an easy??

0 Share Reply

softwareshortcut ★ 436 April 16, 2019 10:51 PM Report

Is there a little mistake in Approach 5? We want to generate a random number from 0 to 61 and not 0 to 62.
alphabet.charAt(62) will break with *StringIndexOutOfBoundsException*. Unless *nextInt()* function excludes the max.

0 Share Reply

SHOW 2 REPLIES

‹ 1 2 ›