Character Encoding

UTF-8, UTF-16, & UTF-32

wuriyan.to

Unicode

Unicode adalah standar teknologi informasi yang dibuat dan disetujui untuk *encoding*, manipulasi, menampilkan, mengirim dan menangani sistem text atau tulisan pada Komputer yang sudah disetujui di seluruh dunia. Standar tersebut dikelola oleh organisasi non profit yang diberi nama **Unicode Consortium**. Anggota dari **Unicode Consortium** sendiri datang dari berbagai kalangan, seperti perusahaan teknologi (seperti Google, Microsoft dll), lembaga pendidikan, Negara, dan perorangan.

Jika anda pernah mendengar **ASCII**, **Unicode** sendiri bisa dikatakan adalah superset dari **ASCII**. Jadi semua character list yang berada pada **ASCII** akan otomatis didukung pada sistem pengkodean **Unicode**. **ASCII** *character set* bisa memenuhi kebutuhan untuk tulisan dan bahasa yang menggunakan *Latin Alphabet*. Tapi untuk memenuhi kebutuhan semua huruf, karakter, emoji dan simbol dari seluruh dunia ASCII tidaklah cukup misalnya huruf Arab, Mandarin, Korea, Rusia, Jawa dan sebagainya. Itulah dasar dibentuknya **Unicode Consortium** untuk menyelesaikan masalah tersebut.

Jika anda tertarik mengetahui lebih detail, saya pernah menulisnya pada artikel berikut:

http://teknologipedia.id/apa-itu-byte-dan-bit-dalam-komputer-dan-apa-hubungannya-dengan-pixel-dan-ascii/

Codepoint

Unicode Character Set-nya sendiri saat ini sudah lebih dari **144.000+**, dan **1,112,064+** kemungkinan, dan akan terus bertambah. Setiap set character-nya diwakili dengan ***index*** yang sudah disetujui pada semua sistem Komputer yang disebut dengan **Codepoint**. Misalnya huruf **H** memiliki **codepoint 72**, jadi pada Komputer, Ponsel, Device, Embedded system manapun codepoint dari huruf H akan tetap **72**.

Contoh karakter **H = 72**

i = 105

🌹 **= 127801**

Anda mungkin pernah berpikir, seperti apakah representasi ***block of memory*** dari text **Hi** 🌹 ketika dikirim melewati jaringan Komputer, disimpan, dan diproses oleh Komputer. **Character Encoding**-lah yang akan membantu Komputer untuk proses manipulasi, mengirim, dan menampilkan text tersebut.

UTF-8

UTF-8 adalah *character encoding* yang menggunakan **8-bit (1 bytes)** *code units* untuk merepresentasikan **Unicode** *code point*. Seperti halnya dengan **UTF-16** (yang akan kita bahas selanjutnya), **UTF-8** juga akan membutuhkan *multiple bytes/multiple code unit* untuk merepresentasikan sebuah **Unicode codepoint**, ketika rule pertama tidak terpenuhi. Yaitu jika *code point* **>= 128**. Jika Unicode codepoint **< 128** maka akan memiliki ukuran 1 bytes.

Bagaimana UTF-8 Encoding bekerja

Algoritma:

- Untuk code point kurang dari **< 128 (2^7-1) = 7 bit**, cukup genapkan menjadi **8 bit**

  contoh karakter **W** = **1010111 (decimal = 87)** *pad* dengan **0** sampai menjadi **8 bit**, sehingga menjadi **01010111**

  contoh karakter * = **101010 (decimal = 42)** *pad* dengan **0** sampai menjadi **8 bit**, sehingga menjadi **00101010**

  Sehingga representasi dalam block of memory untuk karakter W = 01010111

  Sehingga representasi dalam block of memory untuk karakter * = 00101010

Bagaimana UTF-8 Encoding bekerja

- Untuk code point yang lebih dari **>= 128 dan < 2048**, genapkan menjadi **11 bit**

  Ambil **5 bit** pertama dan *pad* dengan **110, bit 110** tersebut melambangkan berapa jumlah **bytes** yang berada dalam *sequence*, pada **bit 110** terdapat **2 bit** bernilai **ON (1)**, yang berarti ada ***2 sequence bytes*** yang merepresentasikan satu **codepoint**.

  Ambil **6 bit** kedua dan *pad* dengan **bit 10**, **bit 10** tersebut melambangkan **continuation bit**, yang berarti **bytes** pada posisi ini adalah bagian dari **bytes** sebelumnya.

  Contoh karakter **Σ = 110101001 (decimal 425)** *pad* dengan **0** sampai menjadi **11 bit**, sehingga menjadi **00110101001**

  Ambil **5 bit** pertama, **00110 | 101001** dan *pad* dengan **110** sehingga menjadi **11000110**

  Ambil **6 bit** kedua, **00110 | 101001** dan *pad* dengan **10** sehingga menjadi **10101001**

  Sehingga representasi dalam block of memory = 11000110 10101001

Bagaimana UTF-8 Encoding bekerja

- Untuk code point yang lebih dari **>= 2048 dan < 65.536**, genapkan menjadi **16 bit**

  Ambil **4 bit** pertama dan *pad* dengan **1110, 1110** berarti ada **(3 sequence bytes)**

  Ambil **6 bit** kedua dan *pad* dengan **10, continuation bit**

  Ambil **6 bit** ketiga dan *pad* dengan **10, continuation bit**

  Contoh karakter ✪ = **10011100101010 (decimal 10026)** *pad* dengan **0** sampai menjadi **16 bit**, sehingga menjadi **0010011100101010**

  Ambil **4 bit** pertama, **0010 | 011100101010** dan *pad* dengan **1110** sehingga menjadi **11100010**

  Ambil **6 bit** kedua, **0010 | 011100 | 101010** dan *pad* dengan **10** sehingga menjadi **10011100**

  Ambil **6 bit** ketiga, **0010 | 011100 | 101010** dan *pad* dengan **10** sehingga menjadi **10101010**

  Sehingga representasi dalam block of memory = 11100010 10011100 10101010

Bagaimana UTF-8 Encoding bekerja

- Untuk code point yang lebih dari **>= 65.536 dan < 1.114.112**, genapkan menjadi **21 bit**

  Ambil **3 bit** pertama dan *pad* dengan **11110, 11110** berarti ada **(4 sequence bytes)**

  Ambil **6 bit** kedua dan *pad* dengan **10, continuation bit**

  Ambil **6 bit** ketiga dan *pad* dengan **10, continuation bit**

  Ambil **6 bit** keempat dan *pad* dengan **10, continuation bit**

  Contoh **Rose Emoji** 🌹 **= 11111001100111001 (decimal 127801)** *pad* dengan **0** sampai menjadi **21 bit**, sehingga menjadi **0000**11111001100111001

  Ambil **3 bit** pertama, **000** | **011111001100111001** dan *pad* dengan **11110** sehingga menjadi **11110**000

  Ambil **6 bit** kedua, **000** | **011111** | **001100111001** dan *pad* dengan **10** sehingga menjadi **10**011111

  Ambil **6 bit** ketiga, **000 | 011111 | 001100 | 111001** dan *pad* dengan **10** sehingga menjadi **10**001100

  Ambil **6 bit** keempat, **000 | 011111 | 001100 | 111001** dan *pad* dengan **10** sehingga menjadi **10**111001

  Sehingga representasi dalam block of memory = 11110000 10011111 10001100 10111001

Bagaimana UTF-8 Encoding bekerja

Bentuk *block memory* text **Hi** 🌹 ketika di *encode* dengan menggunakan **UTF-8**

H = 72 (bin 01001000)

i = 105 (bin 01101001)

🌹 **= 127801 (bin 11110000 10011111 10001100 10111001)**

H i 🌹

01001000 01101001 11110000 10011111 10001100 10111001

Implementasi UTF-8

Mungkin banyak yang belum tau, jika tipe data **char** pada bahasa pemrograman **Rust** adalah **32 bit(4 bytes)**. Berbeda dengan **Java** atau **C#** misalnya, yang tipe data char-nya memiliki panjang **16-bit(2 bytes)** saja.

Karena di **Rust char** adalah **valid UTF-8**, jadi kita bisa menggunakan char langsung untuk implementasi ini. Berikut contoh Algoritma **UTF-8** yang diimplementasikan pada **Rust.**

```
const MAX_ONE_BYTE: u32 = 0x80; // 128
const MAX_TWO_BYTE: u32 = 0x800; // 2048
const MAX_THREE_BYTE: u32 = 0x10000; // 65536

const MASK: u32 = 0x3F; // 63 // 00111111
const CONTINUATION_MASK: u32 = 0x80; // 128 // 10000000
const TWO_BYTE_MASK: u32 = 0xC0; // 192 // 11000000
const THREE_BYTE_MASK: u32 = 0xE0; // 224 // 11100000
const FOUR_BYTE_MASK: u32 = 0xF0; // 240 // 11110000

fn encode_utf8(c: char, out: &mut Vec<u8>) -> Result<(), String> {
   let c_decimal: u32 = c as u32;

   if c_decimal < MAX_ONE_BYTE {
       out.push(c_decimal as u8);
       return Ok(());
   }

   if c_decimal < MAX_TWO_BYTE {
       let b_one: u8 = ((c_decimal >> 6) | TWO_BYTE_MASK) as u8;
       let b_two: u8 = ((c_decimal & MASK) | CONTINUATION_MASK) as u8;
       out.push(b_one);
       out.push(b_two);
       return Ok(());
   }

   if c_decimal < MAX_THREE_BYTE {
       let b_one: u8 = ((c_decimal >> 12) | THREE_BYTE_MASK) as u8;
       let b_two: u8 = (((c_decimal >> 6) & MASK) | CONTINUATION_MASK) as u8;
       let b_three: u8 = ((c_decimal & MASK) | CONTINUATION_MASK) as u8;
       out.push(b_one);
       out.push(b_two);
       out.push(b_three);
       return Ok(());
   }
```

```
    let b_one: u8 = ((c_decimal >> 18) | FOUR_BYTE_MASK) as u8;
    let b_two: u8 = (((c_decimal >> 12) & MASK) | CONTINUATION_MASK) as u8;
    let b_three: u8 = (((c_decimal >> 6) & MASK) | CONTINUATION_MASK) as u8;
    let b_four: u8 = ((c_decimal & MASK) | CONTINUATION_MASK) as u8;
    out.push(b_one);
    out.push(b_two);
    out.push(b_three);
    out.push(b_four);

    Ok(())
}

fn main() {
    let sigma = 'Σ';
    let star = '✪';
    let rose_emoji = '🌹';

    let mut res: Vec<u8> = Vec::new();

    if let Err(e) = encode_utf8(rose_emoji, &mut res) {
        println!("{}", e);
        std::process::exit(1);
    }

    println!("{:?}", res);
}
```

Source code bisa diambil di gist milik saya: https://gist.github.com/wuriyanto48/b28763082123831b7a7cc66a9686d1b7

UTF-16

UTF-16 adalah *character encoding* yang menggunakan **16-bit** *code unit* untuk merepresentasikan **Unicode** *code point*. Satu **16-bit** *code unit* dapat merepresentasikan *code point* **0 - 65.535** atau **2^16-1**.

Untuk merepresentasikan **Unicode** code point diatas **65.535** atau code point yang melebihi panjang **16-bit**, sehingga membutuhkan **2x16-bit** *block of memory*, **UTF-16** menggunakan teknik yang disebut dengan **Surrogate Pairs**. **Surrogate Pairs** atau **Surrogate Code points** adalah *code point* yang di ***reserved*** oleh **UTF-16** dan tidak bisa digunakan untuk merepresentasikan *code point* apapun. Surrogate Pairs memiliki range **U+D800 to U+DFFF (decimal 55.296 to 57.343)**.

Surrogate Pairs code point tersebut dibagi menjadi 2, yaitu High dan Low Surrogate.

High Surrogate 55.296 - 56.319 U+D800 - U+DBFF	Low Surrogate 56.320 - 57.343 U+DC00 - U+DFFF

Bagaimana UTF-16 Encoding bekerja

Kita akan ***encode* Rose Emoji** dengan menggunakan **UTF-16** () . Supaya kita bisa membayangkan bagaimana bentuk representasi data dari **Rose Emoji** ketika disimpan didalam *disc* ataupun ketika data dikirim melewati *Network*.

Berdasarkan **Unicode Consortium**, **Rose Emoji** memiliki *code point* = **127801** dalam *decimal*. Dan berikut representasi dalam **Hexadecimal** dan **Binary**.

- 1F339 (hex)
- 11111001100111001 (bin)

Algoritma:

Berikut Algoritma yang digunakan oleh **UTF-16** untuk meng-*encode* sebuah *character* atau *emoji* misalnya **Rose Emoji** () yang memiliki panjang lebih dari **16-bit**.

1. Kurangi **127.801 dengan 65.536**

   127.801 - 65.536 = 62.265

2. Konversi hasil diatas menjadi format binary **62.265 = 1111001100111001**

Bagaimana UTF-16 Encoding bekerja

3. Buat menjadi 20 digits = **00001111001100111001**
4. Pisah menjadi 10 digits masing masing **0000111100 | 1100111001**
5. Ubah 2 *sequence binary* tersebut menjadi format *decimal*.

   0000111100 = 60

   1100111001 = 825
6. Tambahkan setiap format *decimal*-nya dengan angka awal dari **High Surrogate (55.296)** dan angka awal dari **Low Surrogate (56.320)**

   55.296 + 60 = 55.356

   56.320 + 825 = 57.145

Bagaimana UTF-16 Encoding bekerja

55.296 + 60 = 55.356

56.320 + 825 = 57.145

Sehingga kita dapatkan hasil

High Surrogate = 55.356 (bin 1101100000111100)

Low Surrogate = 57.145 (bin 1101111100111001)

Berikut adalah representasi **Rose Emoji** dalam *binary* ketika data berada pada *block of memory* ataupun ketika data dikirim melewati *Network* jika kita *encode* dengan menggunakan **UTF-16 encoding**.

UTF-16 Encode(🌹) = **11011000001111001101111100111001**

1101100000111100

1101111100111001

Bagaimana UTF-16 Encoding bekerja

Untuk proses **Decoding UTF-16** kita akan ***reverse*** proses sebelumnya untuk mendapatkan code point

dari Rose Emoji (🌹) .

Berikut Formulanya:

code point = 65.536 + ((high surrogate code point - 55.296) * 1024) + (low surrogate code point - 56.320)

65536 + ((55.356 - 55.296) * 1024) + (57.145 - 56.320) = 127.801

Sehingga kita dapatkan hasil **127.801** dimana angka tersebut adalah *code point* dari **Rose Emoji** (🌹).

Bagaimana UTF-16 Encoding bekerja

Bentuk *block memory* text **Hi** 🌹 ketika di *encode* dengan menggunakan **UTF-16.** Pada **UTF-16** sebuah code point bisa memiliki ukuran **2 - 4 bytes**, sebagai contoh **Rose Emoji** (🌹) yang membutuhkan ukuran sampai **4 bytes**.

H = 72 (bin 0000000001001000)

i = 105 (bin 0000000001101001)

🌹 **= 127801 (bin 1101100000111100 1101111100111001)**

H **i** 🌹

0000000001001000 0000000001101001 1101100000111100 1101111100111001

Implementasi UTF-16

Pada bahasa pemrograman **Java**, **C#** dan **Javascript** menggunakan **UTF-16** untuk meng-*encode* **String** dan **Char**. Itulah kenapa tipe data **char** pada **Java** dan **C#** memiliki panjang **16-bit** atau setara dengan **2 bytes**. Sehingga secara logika ketika anda menggunakan **char** pada **Java** dan **C#** maka dia sebenarnya adalah ***sequence*** dari (2x) **16-bit** data. Sedangkan tipe data **String** sendiri dia sebenarnya adalah ***sequence*** dari char, sehingga secara logika String adalah ***sequence*** atau kumpulan dari data yang masing-masing memiliki panjang **16-bit**.

UTF-32 (UCS 4)

UTF-32 adalah *character encoding* yang menggunakan **32-bit (4 bytes)** *code units* untuk merepresentasikan **Unicode** *code point*. UTF-32 memiliki f*ixed byte size* untuk setiap **codepoint**. Walaupun begitu, dengan ukuran **32-bit**, **UTF-32** mampu melakukan *encoding* untuk setiap kemungkinan **Unicode codepoint**. Tidak seperti UTF-8 dan UTF-16 yang membutuhkan formula dan algoritma khusus ketika proses *encoding* dan *decoding*-nya, **UTF-32** merepresentasikan setiap **codepoint** dalam bentuk representasi *binary* asli. Hanya saja setiap *chunk bytes*-nya mempunyai ukuran fix, yaitu **32 bit**.

Bagaimana UTF-32 Encoding bekerja

Bentuk *block memory* text **Hi** 🌹 ketika di *encode* dengan menggunakan **UTF-32**

H = 72 (bin 00000000000000000000000001001000)

i = 105 (bin 00000000000000000000000001101001)

🌹 **= 127801 (bin 00000000000000011111001100111001)**

H **i** 🌹

00000000000000000000000001001000 00000000000000000000000001101001
00000000000000011111001100111001

Kesimpulan

Jika anda perhatikan, **UTF-16** dan **UTF-32** membutuhkan banyak *space memory* dalam implementasinya. Bahkan hanya untuk karakter seperti **“H”** yang sebenarnya hanya membutuhkan paling banyak **8 bit** *block memory*. Jadi usahakan selalu menggunakan **UTF-8**, jika tidak ada kebutuhan khusus yang mengharuskan menggunakan **UTF-16** atau **UTF-32**.

Saat ini **UTF-8** juga menjadi *de facto* pada setiap Sistem operasi, bahasa pemrograman, dan lebih dari 95% aplikasi web di seluruh dunia.

Memiliki pemahaman pada Character Encoding juga sangat penting bagi seorang programmer, ketika anda memahaminya dengan baik, anda sudah memenuhi salah satu skill fundamental seorang programmer. Anda akan merasa mudah ketika sudah mulai terjun ke *low level programming*, misalnya *low level network programming*, mempelajari protocol (Http, Websocket, Database protocol), dan IOT.

Atau anda ingin membuat sistem *chat* yang setiap karakternya menggunakan Aksara **Jawa** atau **Sunda** misalnya. Untuk kebutuhan seperti ini anda akan membutuhkan pemahaman yang kuat terhadap **Unicode** dan **Character Encoding**.