문자인코딩

아스키 코드, 유니코드 그리고 UTF-8, UTF-16

사용자 dingue | 2019. 6. 17. 17:41

인코딩

- 부호화나 인코딩은 정보의 형태나 형식을 변환하는 처리나 처리방식
- 문자 인코딩은 문자들의 집합을 부호화 하는 방식

→ 어떤 정보를 미리 약속한 규칙으로 가공하는 것입니다.

문자 인코딩

문자를 전자화 하기 위해서 각 문자를 추상적인 숫자와 짝지어 놓고, 그 숫자를 컴퓨터 안에서 저장 처리하기 위해서 컴퓨터가 이해할 수 있도록 숫자를 표현한 방식입니다.

ex) 아스키 코드

이진법	팔진법	십진법	십육진법	모양	85진법 (아스키 85)
100 0000	100	64	40	@	31
100 0001	101	65	41	A	32
100 0010	102	66	42	B	33
100 0011	103	67	43	C	34
100 0100	104	68	44	D	35
100 0101	105	69	45	E	36
100 0110	106	70	46	F	37
100 0111	107	71	47	G	38

A, B라는 문자를 100 0001, 100 0010 처럼 컴퓨터가 이해할 수 있는 숫자로 표현

아스키 코드

→ ASCII (American Standard Code for Information Interchange, 미국 정보 교환 표준 부호)

초창기의 컴퓨터는 사람과 기계어로만 소통을 했었고, 당연히 사용 과정에 상당한 애로사항을 겪었다. 그래서 어셈블리어 등의 사람이 쓰는 문자를 사용할 필요성이 생겼기 때문에, 라틴 문자와 숫자, 몇몇 특수문자를 128개(2^7, 27)의 코드값에 1:1 대응시키는 법을 고안했고, 이것이 아스키코드입니다.

초기에는 8비트 중에서 7비트는 문자를 표현하고 1비트는 통신 에러를 감지하기 위한 체크섬으로 사용했으나, 현재는 체크섬이 필요없어지면서, 문자를 표현한 7비트 앞에 0을 집어넣는 방식으로 사용하고 있습니다.

아래 링크로 들어가서 표를 보시면 아스키 코드가 문자를 어떤 방식으로 표현했는지 보실 수 있습니다.

https://ko.wikipedia.org/wiki/ASCII#%EC%B6%9C%EB%A0%A5_%EA%B0%80%EB%8A%A5_%EC%95%84%EC%8A%A4%ED%8

이름에서도 알 수 있듯이 미국에서 만들었으며, 영문자만을 표현합니다. 그렇기 때문에 유럽, 아랍, 아시아 등에서는 아스키 코드만으로는 컴퓨터에서 문자를 표현할 수 없었습니다.

그래서 각자 아스키 코드를 확장해서 자신만의 문자 인코딩을 만들었으며, 각자 자신의 문자 인코딩을 사용하게 되면서 통신할 때 문자가 깨지는 현상이 발생했습니다.

그래서 나온것이 유니코드입니다.

유니코드

전 세계의 문자를 컴퓨터에서 일관되게 표현하고 다룰 수 있도록 설계된 산업 표준입니다.

ASCII가 1바이트에 문자를 표현했던 것과 달리 최대 21비트에 전세계의 문자를 표현합니다.

현재의 유니코드는 지구상에서 통용되는 대부분의 문자들을 담고 있습니다. 여기에는 언어를 표기할 때 쓰는 문자는 물론, 악보 기호, 이모티콘 ʕ•ᴥ•ʔ, 태그, 마작이나 도미노 기호, 𝒰𝓃𝒾𝒸𝑜𝒹𝑒 같은 것들도 포함됩니다.

문자배정 방식

초기에 유니코드는 0x0000부터 0xFFFF 까지 2^16에 모든 문자를 집어넣었습니다.

하지만 2^16개의 자리로도 부족해지자 0x0000부터 0xFFFF까지 문자를 배정해놓은 세트를 총 17개로 늘리게 됩니다.

이 세트들을 유니코드에서는 "평면"이라고 부르고, 앞에서부터 0번 ~ 16번 까지 번호를 붙여놓았습니다. (현재 사용되고 있는 평면은 0~2번과 14~16번의 6개 평면입니다.)

이 평면중에서 현대에서 쓰이는 문자들을 모두 0번 평면에 몰아넣었는데 이 평면을 기본 다국어 평면(BMP, Basic multilingual plane)라고 합니다.

유니코드를 표시하는 방법 : U+

유니코드에서는 모든 평면이 0x0000부터 0xFFFF의 값을 가지고 있기 때문에 문자끼리 겹치는 현상이 발생했습니다.

예를 들어서 0xB000은 0번 평면에서는 한글 "꿰"이지만 1번 평면에서는

라는 글자가 됩니다.

이런 문제를 해결하기 위해서 유니코드에서는 U+ 라는 기호에 평면 숫자를 붙이는 방식을 사용했습니다. (단, 0번 평면의 경우는 숫자를 붙이는 것을 생략했습니다.

ex) 한글 "꿰"은 0번 평면에 있으니 아무것도 붙이지 않고 U+B000라고 표시하면 됩니다.

ex2)

유니코드 인코딩 종류

유니코드로는 세상에 존재하는 거의 모든 문자를 표현할 수 있지만, 그만큼 바이트를 많이 사용하기 때문에 용량이 크다는 문제가 있습니다.

또, 첫 번째로 평면을 표시하는 숫자를 앞에 붙여야 하기 때문에 문자를 표시하는 바이트 외에 자리가 더 필요한 상황이 됩니다.

그런 문제를 해결하기 위해서 유니코드는 많은 인코딩 방식을 가지고 있습니다.

유니코드의 인코딩 방식으로는 UCS-2, UCS-4, 그리고 UTF-8, UTF-16, UTF-32 등이 있습니다.

UCS-2

UCS-2 방식은 유니코드의 모든 평면의 문자를 모두 인코딩 하지 않고, 0번 기본 다국어 평면 (BMP)만을 인코딩하는 방식입니다.

0번 평면의 0x0000~0xFFFF까지의 문자를 그대로 고정 2바이트 형식으로 표현합니다.

ex) 유니코드에서의 한글 "가"에 해당하는 16진수 숫자는 0xAC00을 UCS-2로 인코딩 하면 [0xAC 0x00]이 됩니다.

UCS-2의 장점

유니코드에서 평면을 나타내는 숫자를 떼어버렸기 때문에, 1글자의 크기가 딱 2바이트로 깔끔하게 떨어집니다.

UCS-2의 단점

1. 2바이트를 사용해서 문자를 표현하기 때문에 엔디안 문제가 발생합니다.

ex) 0xAC00은 빅엔디안을 사용하는 시스템에서는 한글 "가"가 되지만, 리틀 엔디안을 사용하는 시스템에서는 ¬ 라는 제어문자가 됩니다.

이를 해결하기 위해서 2바이트인 BOM(Byte Order Mark)을 유니코드 파일이 시작되는 첫 부분에 명시해서 문자열이 어떤 엔디안 방식을 사용하는지 명시하도록 했습니다.

BOM의 종류

인코딩 방식	Byte Order Mark(BOM)
UTF-8	EF BB BF
UTF-16 Big Endian	FE FF
UTF-16 Little Endian	FF FE
UTF-32 Big Endian	00 00 FE FF
UTF-32 Little Endian	FF FE 00 00

ex) 한글 "가"는 BOM이 0xFEFF라면(빅엔디안) 0x0048이 되고 BOM이 0xFFFE라면(리틀엔디안) 0x4800으로 저장됩니다.

하지만 또, 문제가 되는 것은 UCS-2방식의 문자열들이 모두 BOM을 달고 있지는 않다는 점입니다. 이 것 때문에 다른 바이트 순서를 사용하는 시스템에서 문제가 생길 수 있습니다.

2. 0번 평면의 문자들만 인코딩 했기 때문에 모든 유니코드 문자를 표현할 수 없습니다.

UTF-32

기본 다국어 평면(BMP)만을 이용한 UCS-2와는 달리 UTF-32는 유니코드의 모든 문자를 표현하기 때문에 한 글자당 32비트를 사용하는 인코딩입니다.

표현 방식

UTF-32에서 앞에 2바이트는 [0x00 0x00]부터 [0x00 0x10]까지 몇 번째 평면인가를 표시합니다. 그리고 뒤에 2바이트는 UCS-2와 같이 해당 평면의 어느 문자인지를 나타냅니다.

ex)

UTF-32의 장점

유니코드를 그대로 4바이트 자리에 가져다 놓았기 때문에 유니코드와 비교하기 쉽고, 한 글자에 고정 4바이트를 할당하기 때문에 문자열 처리가 간결해집니다.

UTF-32의 단점

1. 한 글자에 고정적으로 4바이트를 할당하기 때문에 다른 인코딩 방식에 비해서 많은 용량을 차지하고 있습니다.

ex) 같은 문자라도 ASCII 코드와 비교해서 용량이 4배나 차이가 나게 됩니다.

2. HTML5에서 사용 금지

→ HTML5.1 사양 4.2.5.5 챕터에는 아래와 같이 명시되있습니다.

> *Authors should not use UTF-32, as the encoding detection algorithms described in this specification intentionally do not distinguish it from UTF-16.*
>
> *저작자는 UTF-32를 사용해서는 안된다. 본 사양에서 서술한 인코딩 감지 알고리즘은 UTF-32를 UTF-16과 의도적으로 구분하지 않기 때문이다.*

이런 문제들 때문에 UTF-32은 거의 사용되지 않고 있습니다.

UTF-16

UTF-16은 너무 많은 용량을 사용하는 UTF-32 인코딩의 문제를 해결하기 위해 나온 가변 길이 인코딩 기법입니다.

UTF-16은 기본 다국어 평면에 속하는 문자들을 표현할 때는 UCS-2와 같지만, 2바이트를 넘어서는 문자는 4바이트를 표현하도록 하여 문제를 해결했습니다.

문자 표현

2바이트 : 기본 다국어 평면(BMP)
4바이트 : 2바이트를 넘어선 모든 유니코드 문자

ex1) 영문 소문자 z는 기본 다국어 평면의 0x007A 자리에 배정되어 있으므로, UTF-16에서는 [0x00 0x7A]로 표시하게 됩니다.

ex2) 한글 "가"는 기본 다국어 평면의 0xAC00 자리에 배정되어 있기 때문에 UTF-16에서는 [0xAC 0x00]으로 표시할 수 있습니다.

기본 다국어 평면를 제외한 문자들은 위의 예제와는 다르게 4바이트로 표시합니다.

그리고 위처럼 간단한 규칙이 아닌 복잡한 규칙을 통해서 문자를 표현하게 됩니다.

ex) 고대 그리스어에서 사용되었던 "Greek LITRA Sign" 라는 문자는 유니코드로는 U+10183이고, UTF-32로는 [0x00 0x01 0x01 0x83]으로 표시하는 문자입니다.

이를 UTF-16으로 표현하기 위해서는 아래와 같은 과정을 거쳐야 합니다.

위에서 문자를 표현한 비트를 3개 부분으로 나누어야 합니다.

1. 2번째 바이트의 뒤에서부터 5비트
2. 3번째 바이트의 앞에서부터 6비트
3. 3번째 바이트의 뒤에서부터 2비트와 4번째 바이트의 8비트 전체

이렇게 나누면 아래와 같이 나누어 지게 됩니다.

여기서 아래와 같은 과정을 통해서 utf-16 인코딩 값으로 변환을 시켜야 합니다.

1. 가장 앞에서부터 110110을 붙입니다.
2. 빨간 부분에서 1을 뺀 4자리의 2진수 값을 붙입니다. (빨간 부분이 00110이라면, 1을 뺀 0101 값을 이곳에 붙이면 됩니다.)
3. 파란 부분을 그대로 가져와 붙입니다.
4. 그 뒤에 110111을 붙입니다.
5. 마지막으로 노란 부분을 그대로 가져와 붙입니다.

과정을 모두 마치면 아래와 같은 결과가 나오게 됩니다.

그러면 왜 이런 복잡한 방식을 사용하게 됐을 까요?

문자열이 정상인지, 기본 다국어 평면의 문자인지 쉽게 확인할 수 있기 때문입니다.

유니코드의 기본 다국어 평면에는 문자가 배정되어 있지 않은 영역이 2군데가 있습니다.

하나는 앞의 6비트가 110110으로 시작하는 영역이고, 다른 하나는 앞의 6비트가 110111로 시작하는 영역입니다.

그렇기 때문에 UTF-16의 앞의 6비트를 확인했을 때 110110이나 110111로 시작하지 않는다면 거기서부터 2바이트는 무조건 기본 다국어 평면 상에 있는 문자라고 확신할 수 있게 됩니다.

만약 반대로 110110으로 시작한다면 기본 다국어 평면 외의 문자라고 단언할 수 있게 됩니다.

위의 110110으로 시작하는 문자나, 110111로 시작하는 문자를 서러게이트(Surrogate) 영역 문자라고 합니다.

하지만 UTF-16은 아래와 같은 문제가 있었습니다.

1. 최소 2바이트를 사용해 문자를 표현하기 때문에 엔디안 문제가 발생합니다.

   → 리틀엔디안인지, 빅엔디안인지에 따라 값이 바뀔 수 있습니다.

2. 아스키 코드와 호환되지 않았습니다.

   → 최소 2바이트를 사용하기 때문에 1바이트를 사용하는 아스키코드랑 호환되지 않습니다.

대표적으로 UTF-16을 사용하는 곳

Java의 String, 윈도우 운영체제의 메모리 저장

UTF-8

UTF-8은 가장 많이 사용되는 가변 길이 인코딩입니다. 문자에 따라서 1바이트 ~4바이트로 인코딩 될 수 있으며, UTF-16과 다르게 아스키 코드와 하위 호환성을 가집니다.

보통 유니코드가 널리 쓰이기 전에 형성된 문서나 프로그램이 아스키 코드를 기반으로 작성되었기 때문에 UTF-8은 이 부분에서 많은 장점을 가지게 되어서 널리 쓰이게 되었습니다.

표현 방식

U+0000~U+007F : UTF-8 Encoding 0xxx xxxx (총 1byte)
U+0080~U+07FF : UTF-8 Encoding 110x xxxx 10xx xxxx (총 2bytes)
U+0800~U+FFFF : UTF-8 Encoding 1110 xxxx 10xx xxxx 10xx xxxx (총 3bytes)
U+10000~U+10FFFF : UTF-8 Encoding 1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx
(총 4bytes)

UTF-8은 위 규칙에 따라서 유니코드 문자를 인코딩 합니다.

1바이트 : ASCII코드

문자가 ASCII 코드인 경우에는 ASCII와 동일하게 1바이트로 표현을 합니다.

2바이트 : 아랍, 히브리, 대부분의 유럽계(조지안 문자 제외)

UTF-8에서는 U+0080 ~ U+07FF 사이의 아랍, 히브리, 유럽계 문자들을 2바이트로 표현합니다.

여기서 앞의 바이트 앞에 110을 붙이고, 따라오는 바이트의 앞에는 10을 붙이는 규칙을 적용합니다.

ex)

아래 그림과 같은 유니코드 문자가 있을 때

UTF-8의 규칙에 따라서 첫 바이트 앞에 110을 붙인 뒤, 빨간색 비트를 붙이고

그 다음 바이트 앞에 10을 붙인 뒤, 나머지 파란색 비트를 붙이는 방식으로 변환합니다.

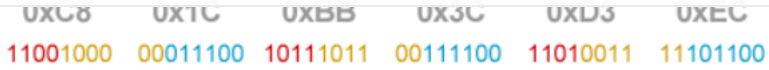
아래는 변환된 그림입니다.

3바이트 : 기본 다국어 평면(BMP)

ASCII와 아랍, 히브리, 유럽계 문자들을 제외한 BMP 문자들은 모두 3바이트로 표현합니다.

3바이트로 표현하는 규칙을 2바이트와 유사하게 적용됩니다.

첫 바이트 앞에는 1110을 붙이고, 나머지 두 바이트 앞에는 10을 붙이고, 문자 비트들을 그 뒤에 적용시키게 됩니다.



위와 같은 문자들이 있을 때

UTF-8은 아래와 같이 변환됩니다.

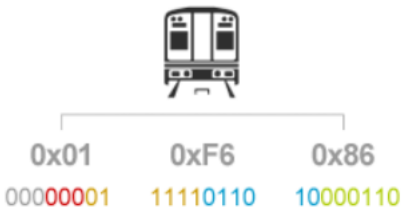
4바이트 : 모든 유니코드 문자

0번 평면인 기본 다국어 평면을 제외한 모든 문자들은 4바이트로 표현합니다.

4바이트로 표현하는 규칙도 앞의 규칙들과 유사합니다.

4바이트중 첫 번째 바이트 앞에 11110을 붙이고, 나머지 바이트 앞에는 10을 붙인 뒤 나머지 문자에 해당하는 비트를 집어넣습니다.

유니코드로 U+1F686인 TRAIN이라는 기호를 예로 보여드리겠습니다.



총 4구역으로 숫자를 쪼개야 합니다.

1. 첫 번째 바이트의 4번째 비트부터 3비트
2. 첫 번째 바이트의 뒤의 2비트와 2번째 바이트의 앞의 4비트
3. 두 번째 바이트 뒤의 4비트와 세번째 바이트 앞의 2비트
4. 세 번째 바이트 뒤의 6비트

쪼갠 숫자를 UTF-8의 규칙에 맞게 각각 아래와 같이 적용시킵니다.

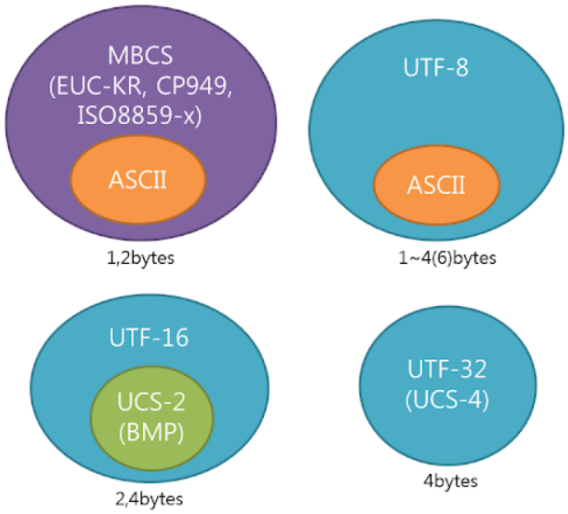
UTF-8에서의 앞의 비트를 이용한 문자 종류 찾기

UTF-8의 적용되는 규칙을 이용해서 이 문자가 어떤 문자인지 유추해낼 수 있습니다.

1. 어떤 바이트가 0으로 시작하면 그 바이트는 ASCII 코드입니다.
2. 어떤 바이트가 110으로 시작하면, 거기서부터 2바이트가 한 글자입니다.
3. 어떤 바이트가 1110으로 시작하면, 거기서부터 3바이트가 한 글자입니다.
4. 어떤 바이트가 11110으로 시작하면, 거기서부터 4바이트가 한 글자입니다.

위와 같은 규칙으로 문자를 유추해낼 수 있습니다.

정리



초창기에 컴퓨터를 문자에서 사용하기 위해서 ASCII 코드가 나왔습니다.

ASCII코드는 한바이트를 이용해서 총 128개 영문권 문자를 나타냈고, 바이트의 맨 왼쪽 비트를 0으로 사용합니다.

이후 ASCII 코드를 확장해서 여러 나라에서 각자 인코딩을 만들어서 사용했고, 그 때 인코딩이 통일되지 않으면서, 문자가 깨지는 현상이 발생했고, 이 문제를 해결하기 위해서 나온 것이 전세계 모든 문자를 숫자로 표현한 유니코드였습니다.

유니코드는 전세계의 문자를 표현했지만, 용량이 크고 문자코드 이외의 어떤 평면에 위치해 있는지에 대한 문제를 해결해야 했기 때문에 여러가지 인코딩이 존재합니다.

UCS-2의 경우에는 고정 2바이트로 유니코드에 있는 문자를 표현했지만, ASCII와 호환이 되지 않았고, 모든 유니코드 문자를 표현할 수 없었습니다.

그래서 UTF-16이 나왔습니다. 모든 문자에 4바이트를 할당하지 않고, 2바이트로 표현할 수 있으면 2바이트로 표현해서 용량문제는 어느정도 해결했지만, 역시 ASCII와 호환되지 않았습니다.

마지막으로 UTF-8이 나왔습니다. UTF-8은 문자 종류에 따라서 1바이트 부터 4바이트 까지 문자를 나타냈습니다.

기본 다국어 평면의 문자들은 UTF-16보다 1바이트를 더 사용하지만, ASCII코드와 완벽하게 호환이 되기 때문에 오늘날 널리 사용하고 있는 인코딩 방식입니다.

출처

https://pickykang.tistory.com/13 - utf-8, utf-16 차이 (삶, 금융, IT, 잡동사니)
https://namu.wiki/w/%EC%9D%B8%EC%BD%94%EB%94%A9 - 인코딩(나무위키)
https://d2.naver.com/helloworld/19187 - 한글 인코딩의 이해 1편: 한글 인코딩의 역사와 유니코드 (네이버 d2)
https://d2.naver.com/helloworld/76650 - 유니코드와 Java를 이용한 한글 처리
https://namu.wiki/w/%EC%95%84%EC%8A%A4%ED%82%A4%20%EC%BD%94%EB%93%9C?from=%EC%95%84%EC%8A%A4%ED%82%A4%EC%BD%94%EB%93%9C - 아스키코드 (나무위키)
https://parkcheolu.tistory.com/4 - 유니코드와 UTF-8, UTF-16 (박철우의 블로그)
https://sleepyeyes.tistory.com/3 - 유니코드와 UTF-8, UTF-16 (졸린 눈)
https://wiki.kldp.org/Translations/html/UTF8-Unicode-KLDP/UTF8-Unicode-KLDP-7.html - UTF-8은 무엇인가? (위키)
https://goodgid.github.io/ASCII-Code/ - 아스키 코드 (goodgid)
http://dev.epiloum.net/595 - 웹개발자를 위한 문자 인코딩 기초 #7 - UTF-16 (Epiloum 개발노트)
http://dev.epiloum.net/164 - 웹개발자를 위한 문자 인코딩 기초#1 - ASCII 코드 (Epiloum 개발노트)
https://brownbears.tistory.com/124 - 유니코드 BOM (Byte Order Mark)

'문자인코딩' 카테고리의 다른 글

아스키 코드, 유니코드 그리고 UTF-8, UTF-16 (0)

링크

TAG more

@NoArgsC... 로그레벨 Val 멀티쓰레드 lombok GC pinpoint Authentic... 메모리 구조 CMS GC 메모리 가시성 성장 개고생 ORM SpringBoo... springboot @nonnull volatile Var gradle @Getter/... Springsec... Java 인터페이스 ... Spring JPA Intellij

« 2020/09 »

일	월	화	수	목	금	토
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

글 보관함

2019/07 (2)
2019/06 (2)
2019/05 (11)
2018/11 (3)