Managing character sets and encodings

There are many languages in use throughout the world, and they use many different character sets. There are also many ways of encoding character sets into binary formats of bytes. This chapter considers some of the issues in this.

Introduction

Once upon a time there was EBCDIC and ASCII... Actually, it was never that simple and has just become more complex over time. There is light on the horizon, but some estimates are that it may be 50 years before we all live in the daylight on this!

Early computers were developed in the english-speaking countries of the US, the UK and Australia. As a result of this, assumptions were made about the language and character sets in use. Basically, the Latin alphabet was used, plus numerals, punctuation characters and a few others. These were then encoded into bytes using ASCII or EBCDIC.

The character-handling mechanisms were based on this: text files and I/O consisted of a sequence of bytes, with each byte representing a single character. String comparison could be done by matching corresponding bytes; conversions from upper to lower case could be done by mapping individual bytes, and so on.

There are about 6,000 living languages in the world (3,000 of them in Papua New Guinea!). A few languages use the "english" characters but most do not. The Romanic languages such as French have adornments on various characters, so that you can write "j'ai arrêté", with two differently accented vowels. Similarly, the Germanic languages have extra characters such as 'ß'. Even UK English has characters not in the standard ASCII set: the pound symbol '£' and recently the euro '€'

But the world is not restricted to variations on the Latin alphabet. Thailand has its own alphabet, with words looking like this: "ภาษาไทย". There are many other alphabets, and Japan even has two, Hiragana and Katagana.

There are also the hierographic languages such as Chinese where you can write "百度一下，你就知道".

It would be nice from a technical viewpoint if the world just used ASCII. However, the trend is in the opposite direction, with more and more users demanding that software use the language that they are familiar with. If you build an application that can be run in different countries then users will demand that it uses their own

language. In a distributed system, different components of the system may be used by users expecting different languages and characters.

Internationalisation (i18n) is how you write your applications so that they can handle the variety of languages and cultures. *Localisation* (l10n) is the process of customising your internationalised application to a particular cultural group.

i18n and l10n are big topics in themselves. For example, they cover issues such as colours: while white means "purity" in Western cultures, it means "death" to the Chinese and "joy" to Egyptians. In this chapter we just look at issues of character handling.

Definitions

It is important to be careful about exactly what part of a text handling system you are talking about. Here is a set of definitions that have proven useful.

Character

A character is a "unit of information that roughly corresponds to a grapheme (written symbol) of a natural language, such as a letter, numeral, or punctuation mark" (Wikipedia). A character is "the smallest component of written language that has a semantic value" (Unicode). This includes letters such as 'a' and 'À' (or letters in any other language), digits such as '2', punctuation characters such as ',' and various symbols such as the English pound currency symbol '£'.

A character is some sort of abstraction of any actual symbol: the character 'a' is to any written 'a' as a Platonic circle is to any actual circle. The concept of character also includes control characters, which do not correspond to natural language symbols but to other bits of information used to process texts of the language.

A character does not have any particular appearance, although we use the appearance to help recognise the character. However, even the appearance may have to be understood in a context: in mathematics, if you see the symbol π (pi) it is the character for the ratio of circumference to radius of a circle, while if you are reading Greek text, it is the sixteenth letter of the alphabet: "προσ" is the greek word for "with" and has nothing to do with 3.14159...

Character repertoire/character set

A character repertoire is a set of distinct characters, such as the Latin alphabet. No particular ordering is assumed. In English, although we say that 'a' is earlier in the

alphabet than 'z', we wouldn't say that 'a' is less than 'z'. The "phone book" ordering which puts "McPhee" before "MacRea" shows that "alphabetic ordering" isn't critical to the characters.

A repertoire specifies the names of the characters and often a sample of how the characters might look. e.g the letter 'a' might look like 'a', '*a*' or '**a**'. But it doesn't force them to look like that - they are just samples. The repertoire may make distinctions such as upper and lower case, so that 'a' and 'A' are different. But it may regard them as the same, just with different sample appearances. (Just like some programming languages treat upper and lower as different - e.g. Go - but some don't e.g. Basic.). On the other hand, a repertoire might contain different characters with the same sample appearance: the repertoire for a Greek mathematician would have two different characters with appearance π. This is also called a noncoded character set.

Character code

A character code is a mapping from characters to integers. The mapping for a character set is also called a coded character set or code set. The value of each character in this mapping is often called a code point. ASCII is a code set. The codepoint for 'a' is 97 and for 'A' is 65 (decimal).

The character code is still an abstraction. It isn't yet what we will see in text files, or in TCP packets. However, it is getting close. as it supplies the mapping from human oriented concepts into numerical ones.

Character encoding

To communicate or store a character you need to encode it in some way. To transmit a string, you need to encode all characters in the string. There are many possible encodings for any code set.

For example, 7-bit ASCII code points can be encoded as themselves into 8-bit bytes (an octet). So ASCII 'A' (with codepoint 65) is encoded as the 8-bit octet 01000001. However, a different encoding would be to use the top bit for parity checking e.g. with odd parity ASCII 'A" would be the octet 11000001. Some protocols such as Sun's XDR use 32-bit word-length encoding. ASCII 'A' would be encoded as 00000000 00000000 0000000 01000001.

The character encoding is where we function at the programming level. Our programs deal with encoded characters. It obviously makes a difference whether we are dealing with 8-bit characters with or without parity checking, or with 32-bit characters.

The encoding extends to strings of characters. A word-length even parity encoding of "ABC" might be 10000000 (parity bit in high byte) 0100000011 (C) 01000010 (B) 01000001 (A in low byte). The comments about the importance of an encoding apply equally strongly to strings, where the rules may be different.

Transport encoding

A character encoding will suffice for handling characters within a single application. However, once you start sending text *between* applications, then there is the further issue of how the bytes, shorts or words are put on the wire. An encoding can be based on space- and hence bandwidth-saving techniques such as **zip**'ping the text. Or it could be reduced to a 7-bit format to allow a parity checking bit, such as **base64**.

If we do know the character and transport encoding, then it is a matter of programming to manage characters and strings. If we don't know the character or transport encoding then it is a matter of guesswork as to what to do with any particular string. There is no convention for files to signal the character encoding.

There *is* however a convention for signalling encoding in text transmitted across the internet. It is simple: the header of a text message contains information about the encoding. For example, an HTTP header can contain lines such as

```
Content-Type: text/html; charset=ISO-8859-4
Content-Encoding: gzip
```

which says that the character set is ISO 8859-4 (corresponding to certain countries in Europe) with the default encoding, but then **gzip**ed. The second part - content encoding - is what we are referring to as "transfer encoding" (IETF RFC 2130).

But how do you read this information? Isn't it encoded? Don't we have a chicken and egg situation? Well, no. The convention is that such information is given in ASCII (to be precise, US ASCII) so that a program can read the headers and then adjust its encoding for the rest of the document.

ASCII

ASCII has the repertoire of the English characters plus digits, punctuation and some control characters. The code points for ASCII are given by the familiar table

```
      Oct   Dec   Hex   Char                Oct   Dec   Hex   Char
      ------------------------------------------------------------
      000   0     00    NUL '\0'            100   64    40    @
      001   1     01    SOH                 101   65    41    A
      002   2     02    STX                 102   66    42    B
```

003	3	03	ETX	103	67	43	C	
004	4	04	EOT	104	68	44	D	
005	5	05	ENQ	105	69	45	E	
006	6	06	ACK	106	70	46	F	
007	7	07	BEL '\a'	107	71	47	G	
010	8	08	BS '\b'	110	72	48	H	
011	9	09	HT '\t'	111	73	49	I	
012	10	0A	LF '\n'	112	74	4A	J	
013	11	0B	VT '\v'	113	75	4B	K	
014	12	0C	FF '\f'	114	76	4C	L	
015	13	0D	CR '\r'	115	77	4D	M	
016	14	0E	SO	116	78	4E	N	
017	15	0F	SI	117	79	4F	O	
020	16	10	DLE	120	80	50	P	
021	17	11	DC1	121	81	51	Q	
022	18	12	DC2	122	82	52	R	
023	19	13	DC3	123	83	53	S	
024	20	14	DC4	124	84	54	T	
025	21	15	NAK	125	85	55	U	
026	22	16	SYN	126	86	56	V	
027	23	17	ETB	127	87	57	W	
030	24	18	CAN	130	88	58	X	
031	25	19	EM	131	89	59	Y	
032	26	1A	SUB	132	90	5A	Z	
033	27	1B	ESC	133	91	5B	[	
034	28	1C	FS	134	92	5C	\	'\\'
035	29	1D	GS	135	93	5D	]	
036	30	1E	RS	136	94	5E	^	
037	31	1F	US	137	95	5F	_	
040	32	20	SPACE	140	96	60	`	
041	33	21	!	141	97	61	a	
042	34	22	"	142	98	62	b	
043	35	23	#	143	99	63	c	
044	36	24	$	144	100	64	d	
045	37	25	%	145	101	65	e	
046	38	26	&	146	102	66	f	
047	39	27	'	147	103	67	g	
050	40	28	(	150	104	68	h	
051	41	29	)	151	105	69	i	
052	42	2A	*	152	106	6A	j	
053	43	2B	+	153	107	6B	k	
054	44	2C	,	154	108	6C	l	
055	45	2D	-	155	109	6D	m	
056	46	2E	.	156	110	6E	n	
057	47	2F	/	157	111	6F	o	
060	48	30	0	160	112	70	p	
061	49	31	1	161	113	71	q	
062	50	32	2	162	114	72	r	
063	51	33	3	163	115	73	s	
064	52	34	4	164	116	74	t	
065	53	35	5	165	117	75	u	
066	54	36	6	166	118	76	v	
067	55	37	7	167	119	77	w	
070	56	38	8	170	120	78	x	
071	57	39	9	171	121	79	y	
072	58	3A	:	172	122	7A	z	
073	59	3B	;	173	123	7B	{	

```
074   60   3C   <              174   124   7C   |
075   61   3D   =              175   125   7D   }
076   62   3E   >              176   126   7E   ~
077   63   3F   ?              177   127   7F   DEL
```

The most common encoding for ASCII uses the code points as 7-bit bytes, so that the encoding of 'A' for example is 65.

This set is actually *US ASCII*. Due to European desires for accented characters, some punctuation characters are omitted to form a minimal set, ISO 646, while there are "national variants" with suitable European characters. The page http://www.cs.tut.fi/~jkorpela/chars.html by Jukka Korpela has more information for those interested. We shall not need these variants though.

ISO 8859

Octets are now the standard size for bytes. This allows 128 extra code points for extensions to ASCII. A number of different code sets to capture the repertoires of various subsets of European languages are the ISO 8859 series. ISO 8859-1 is also known as Latin-1 and covers many languages in western Europe, while others in this series cover the rest of Europe and even Hebrew, Arabic and Thai. For example, ISO 8859-5 includes the Cyrillic characters of countries such as Russia, while ISO 8859-8 includes the Hebrew alphabet.

The standard encoding for these character sets is to use their code point as an 8-bit value. For example, the character 'Á' in ISO 8859-1 has the code point 193 and is encoded as 193. All of the ISO 8859 series have the bottom 128 values identical to ASCII, so that the ASCII characters are the same in all of these sets.

The HTML specifications used to recommend the ISO 8859-1 character set. HTML 3.2 was the last one to do so, and after that HTML 4.0 recommended Unicode. In 2010 Google made an estimate that of the pages it sees, about 20% were still in ISO 8859 format while 20% were still in ASCII ("Unicode nearing 50% of the web" http://googleblog.blogspot.com/2010/01/unicode-nearing-50-of-web.html).

Unicode

Neither ASCII nor ISO 8859 cover the languages based on hieroglyphs. Chinese is estimated to have about 20,000 separate characters, with about 5,000 in common use. These need more than a byte, and typically two bytes has been used. There have been many of these two-byte character sets: Big5, EUC-TW, GB2312 and GBK/GBX for

Chinese, JIS X 0208 for Japanese, and so on. These encodings are generally not mutually compatable.

Unicode is an embracing standard character set intended to cover all major character sets in use. It includes European, Asian, Indian and many more. It is now up to version 5.2 and has over 107,000 characters. The number of code points now exceeds 65,536, that is. more than 2^16. This has implications for character encodings.

The first 256 code points correspond to ISO 8859-1, with US ASCII as the first 128. There is thus a backward compatability with these major character sets, as the code points for ISO 8859-1 and ASCII are exactly the same in Unicode. The same is not true for other character sets: for example, while most of the Big5 characters are also in Unicode, the code points are not the same. The page http://moztw.org/docs/big5/table/unicode1.1-obsolete.txt contains one example of a (large) table mapping from Big5 to Unicode.

To represent Unicode characters in a computer system, an encoding must be used. The encoding UCS is a two-byte encoding using the code point values of the Unicode characters. However, since there are now too many characters in Unicode to fit them all into 2 bytes, this encoding is obsolete and no longer used. Instead there are:

- UTF-32 is a 4-byte encoding, but is not commonly used, and HTML 5 warns explicitly against using it
- UTF-16 encodes the most common characters into 2 bytes with a further 2 bytes for the "overflow", with ASCII and ISO 8859-1 having the usual values
- UTF-8 uses between 1 and 4 bytes per character, with ASCII having the usual values (but not ISO 8859-1)
- UTF-7 is used sometimes, but is not common

UTF-8, Go and runes

UTF-8 is the most commonly used encoding. Google estimates that 50% of the pages that it sees are encoded in UTF-8. The ASCII set has the same encoding values in UTF-8, so a UTF-8 reader can read text consisting of just ASCII characters as well as text from the full Unicode set.

Go uses UTF-8 encoded characters in its strings. Each character is of type `rune`. This is a alias for `int32` as a Unicode character can be 1, 2 or 4 bytes in UTF-8 encoding. In terms of characters, a string is an array of runes.

A string is also an array of bytes, but you have to be careful: only for the ASCII subset is a byte equal to a character. All other characters occupy two, three or four

bytes. This means that the length of a string in characters (runes) is generally not the same as the length of its byte array. They are only equal when the string consists of ASCII characters only.

The following program fragment illustrates this. If we take a UTF-8 string and test its length, you get the length of the underlying byte array. But if you cast the string to an array of runes `[]rune` then you get an array of the Unicode code points which is generally the number of characters:

```
str := "百度一下，你就知道"

println("String length", len([]rune(str)))
println("Byte length", len(str))
```

prints

```
String length 9
Byte length 27
```

UTF-8 client and server

Possibly surprisingly, you need do nothing special to handle UTF-8 text in either the client or the server. The underlying data type for a UTF-8 string in Go is a byte array, and as we saw just above, Go looks after encoding the string into 1, 2, 3 or 4 bytes as needed. The length of the string is the length of the byte array, so you write any UTF-8 string by writing the byte array.

Similarly to read a string, you just read into a byte array and then cast the array to a string using `string([]byte)`. If Go cannot properly decode bytes into Unicode characters, then it gives the Unicode Replacement Character \uFFFD. The length of the resulting byte array is the length of the legal portion of the string.

So the clients and servers given in earlier chapters work perfectly well with UTF-8 encoded text.

ASCII client and server

The ASCII characters have the same encoding in ASCII and in UTF-8. So ordinary UTF-8 character handling works fine for ASCII characters. No special handling need to be done.

UTF-16 and Go

UTF-16 deals with arrays of short 16-bit unsigned integers. The package `utf16` is designed to manage such arrays. To convert a normal Go string, that is a UTF-8 string, into UTF-16, you first extract the code points by coercing it into a `[]rune` and then use `utf16.Encode` to produce an array of type `uint16`.

Similarly, to decode an array of unsigned short UTF-16 values into a Go string, you use `utf16.Decode` to convert it into code points as type `[]rune` and then to a string. The following code fragment illustrates this

```
str := "百度一下，你就知道"

runes := utf16.Encode([]rune(str))
ints := utf16.Decode(runes)

str = string(ints)
```

These type conversions need to be applied by clients or servers as appropriate, to read and write 16-bit short integers, as shown below.

Little-endian and big-endian

Unfortunately, there is a little devil lurking behind UTF-16. It is basically an encoding of characters into 16-bit short integers. The big question is: for each short, how is it written as two bytes? The top one first, or the top one second? Either way is fine, as long as the receiver uses the same convention as the sender.

Unicode has addressed this with a special character known as the BOM (byte order marker). This is a zero-width non-printing character, so you never see it in text. But its value 0xfffe is chosen so that you can tell the byte-order:

- In a big-endian system it is FF FE
- In a little-endian system it is FE FF

Text will sometimes place the BOM as the first character in the text. The reader can then examine these two bytes to determine what endian-ness has been used.

UTF-16 client and server

Using the BOM convention, we can write a server that prepends a BOM and writes a string in UTF-16 as

```
/* UTF16 Server
 */
```

```
package main

import (
        "fmt"
        "net"
        "os"
        "unicode/utf16"
)

const BOM = '\ufffe'

func main() {

        service := "0.0.0.0:1210"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                str := "j'ai arrêté"
                shorts := utf16.Encode([]rune(str))
                writeShorts(conn, shorts)

                conn.Close() // we're finished
        }
}

func writeShorts(conn net.Conn, shorts []uint16) {
        var bytes [2]byte

        // send the BOM as first two bytes
        bytes[0] = BOM >> 8
        bytes[1] = BOM & 255
        _, err := conn.Write(bytes[0:])
        if err != nil {
                return
        }

        for _, v := range shorts {
                bytes[0] = byte(v >> 8)
                bytes[1] = byte(v & 255)

                _, err = conn.Write(bytes[0:])
                if err != nil {
                        return
                }
        }
}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

while a client that reads a byte stream, extracts and examines the BOM and then decodes the rest of the stream is

```
/* UTF16 Client
 */
package main

import (
        "fmt"
        "net"
        "os"
        "unicode/utf16"
)

const BOM = '\ufffe'

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        shorts := readShorts(conn)
        ints := utf16.Decode(shorts)
        str := string(ints)

        fmt.Println(str)

        os.Exit(0)
}

func readShorts(conn net.Conn) []uint16 {
        var buf [512]byte

        // read everything into the buffer
        n, err := conn.Read(buf[0:2])
        for true {
                m, err := conn.Read(buf[n:])
                if m == 0 || err != nil {
                        break
                }
                n += m
        }

        checkError(err)
        var shorts []uint16
        shorts = make([]uint16, n/2)
```

```
        if buf[0] == 0xff && buf[1] == 0xfe {
                // big endian
                for i := 2; i < n; i += 2 {
                        shorts[i/2] = uint16(buf[i])<<8 + uint16(buf[i+1])
                }
        } else if buf[1] == 0xff && buf[0] == 0xfe {
                // little endian
                for i := 2; i < n; i += 2 {
                        shorts[i/2] = uint16(buf[i+1])<<8 + uint16(buf[i])
                }
        } else {
                // unknown byte order
                fmt.Println("Unknown order")
        }
        return shorts

}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Unicode gotcha's

This book is not about i18n issues. In particular we don't want to delve into the arcane areas of Unicode. But you should know that Unicode is not a simple encoding and there are many complexities. For example, some earlier character sets used *non-spacing* characters, particularly for accents. This was brought into Unicode, so you can produce accented characters in two ways: as a single Unicode character, or as a pair of non-spacing accent plus non-accented character. For example, U+04D6 CYRILLIC CAPITAL LETTER IE WITH BREVE is a single character. It is equivalent to U+0415 CYRILLIC CAPITAL LETTER IE combined with the breve accent U+0306 COMBINING BREVE. This makes string comparison difficult on occassions. The Go specification does not at present address such issues.

ISO 8859 and Go

The ISO 8859 series are 8-bit character sets for different parts of Europe and some other areas. They all have the ASCII set common in the low part, but differ in the top part. According to Google, ISO 8859 codes account for about 20% of the web pages it sees.

The first code, ISO 8859-1 or Latin-1, has the first 256 characters in common with Unicode. The encoded value of the Latin-1 characters is the same in UTF-16 and in

the default ISO 8859-1 encoding. But this doesn't really help much, as UTF-16 is a 16-bit encoding and ISO 8859-1 is an 8-bit encoding. UTF-8 is a 8-bit encoding, but it uses the top bit to signal extra bytes, so only the ASCII subset overlaps for UTF-8 and ISO 8859-1. So UTF-8 doesn't help much either.

But the ISO 8859 series don't have any complex issues. To each character in each set corresponds a unique Unicode character. For example, in ISO 8859-2, the character "latin capital letter I with ogonek" has ISO 8859-2 code point 0xc7 (in hexadecimal) and corresponding Unicode code point of U+012E. Transforming either way between an ISO 8859 set and the corresponding Unicode characters is essentially just a table lookup.

The table from ISO 8859 code points to Unicode code points could be done as an array of 256 integers. But many of these will have the same value as the index. So we just use a map of the different ones, and those not in the map take the index value.

For ISO 8859-2 a portion of the map is

```
var unicodeToISOMap = map[int] uint8 {
    0x12e: 0xc7,
    0x10c: 0xc8,
    0x118: 0xca,
    // plus more
}
```

and a function to convert UTF-8 strings to an array of ISO 8859-2 bytes is

```
/* Turn a UTF-8 string into an ISO 8859 encoded byte array
*/
func unicodeStrToISO(str string) []byte {
        // get the unicode code points
        codePoints := []int(str)

        // create a byte array of the same length
        bytes := make([]byte, len(codePoints))

        for n, v := range(codePoints) {
                // see if the point is in the exception map
                iso, ok := unicodeToISOMap[v]
                if !ok {
                        // just use the value
                        iso = uint8(v)
                }
                bytes[n] = iso
        }
        return bytes
}
```

In a similar way you cacn change an array of ISO 8859-2 bytes into a UTF-8 string:

```
var isoToUnicodeMap = map[uint8] int {
    0xc7: 0x12e,
    0xc8: 0x10c,
    0xca: 0x118,
    // and more
}

func isoBytesToUnicode(bytes []byte) string {
        codePoints := make([]int, len(bytes))
        for n, v := range(bytes) {
                unicode, ok :=isoToUnicodeMap[v]
                if !ok {
                        unicode = int(v)
                }
                codePoints[n] = unicode
        }
        return string(codePoints)
}
```

These functions can be used to read and write UTF-8 strings as ISO 8859-2 bytes. By changing the mapping table, you can cover the other ISO 8859 codes. Latin-1, or ISO 8859-1, is a special case - the exception map is empty as the code points for Latin-1 are the same in Unicode. You could also use the same technique for other character sets based on a table mapping, such as Windows 1252.

Other character sets and Go

There are very, very many character set encodings. According to Google, these generally only have a small use, which will hopefully decrease even further in time. But if your software wants to capture all markets, then you may need to handle them.

In the simplest cases, a lookup table will suffice. But that doesn't always work. The character coding ISO 2022 minimised character set sizes by using a finite state machine to swap code pages in and out. This was borrowed by some of the Japanese encodings, and makes things very complex.

Go does not at present give any language or package support for these other character sets. So you either avoid their use, fail to talk to applications that do use them, or write lots of your own code!

Conclusion

There hasn't been much code in this chapter. Instead, there have been some of the concepts of a very complex area. It's up to you: if you want to assume everyone

speaks US English then the world is simple. But if you want your applications to be usable by the rest of the world, then you need to pay attention to these complexities.

Source: http://jan.newmarch.name/go/charsets/chapter-charsets.html