Internationalization

Things about locale, Unicode, gettext, etc.

陈宇飞

Email: cyfdecyf@gmail.com

南京大学

2007 年 3 月 25 日

What I will talk about

- locale 环境变量对系统有什么影响
- 怎样在程序中使用 locale 信息
- 什么是 Unicode, 它和 UTF-8, UTF-16 是什么关系
- 怎样使你的程序使用 Unicode 从而支持中文
- 怎样使程序在不同的语言环境下以不同的语言输出消息

What I will talk about

- locale 环境变量对系统有什么影响
- 怎样在程序中使用 locale 信息
- 什么是 Unicode, 它和 UTF-8, UTF-16 是什么关系
- 怎样使你的程序使用 Unicode 从而支持中文
- 怎样使程序在不同的语言环境下以不同的语言输出消息

What I will talk about

- locale 环境变量对系统有什么影响
- 怎样在程序中使用 locale 信息
- 什么是 Unicode, 它和 UTF-8, UTF-16 是什么关系
- 怎样使你的程序使用 Unicode 从而支持中文
- 怎样使程序在不同的语言环境下以不同的语言输出消息

What I will talk about

- locale 环境变量对系统有什么影响
- 怎样在程序中使用 locale 信息
- 什么是 Unicode, 它和 UTF-8, UTF-16 是什么关系
- 怎样使你的程序使用 Unicode 从而支持中文
- 怎样使程序在不同的语言环境下以不同的语言输出消息

What I will talk about

- locale 环境变量对系统有什么影响
- 怎样在程序中使用 locale 信息
- 什么是 Unicode, 它和 UTF-8, UTF-16 是什么关系
- 怎样使你的程序使用 Unicode 从而支持中文
- 怎样使程序在不同的语言环境下以不同的语言输出消息

Outline

Outline

THE CENTRAL CONCEPT ABOUT INTERNATIONALIZATION

- locale — The place in which a program is run. It encapsulates the following information:
  - local character set
  - how to format and display monetary amounts
  - how to format numeric values
- 一些 locale 值: C, en_US, zh_CN, zh_CN.GBK

OTHERS TERMS ABOUT INTERNATIONALIZATION

- Internationalization (i18n)
  The process of writing (or modifying) a program so that it can function in multiple locales.
- Localization (l10n)
  The process of tailoring an i18n program for a specific locale.
- Globalization (g11n)
  Prepare all possible localizations for an i18n program. Make it for global use.

OTHERS TERMS ABOUT INTERNATIONALIZATION

- Internationalization (i18n)
  The process of writing (or modifying) a program so that it can function in multiple locales.
- Localization (l10n)
  The process of tailoring an i18n program for a specific locale.
- Globalization (g11n)
  Prepare all possible localizations for an i18n program. Make it for global use.

OTHERS TERMS ABOUT INTERNATIONALIZATION

- Internationalization (i18n)
  The process of writing (or modifying) a program so that it can function in multiple locales.
- Localization (l10n)
  The process of tailoring an i18n program for a specific locale.
- Globalization (g11n)
  Prepare all possible localizations for an i18n program. Make it for global use.

关于字符集的三个基本的概念

- Character Set — 字符集
  字符与整数之间的映射
  如果字符集的定义中每一个字符使用超过 8 bit 的整数，则该字符集被称为 Multibyte Character Set
- Character set encoding — 字符集编码
  将字符集中整数转换到其他形式以在计算机中表示
  对字符集中整数值进行编码的目的是为了更好的保存和传输字符
- Language
  定义字符集中字符的使用规则
  例如：字符对应的大写或小写形式，字符的顺序等

关于字符集的三个基本的概念

- Character Set — 字符集
  字符与整数之间的映射
  如果字符集的定义中每一个字符使用超过 8 bit 的整数，则该字符集被称为 Multibyte Character Set
- Character set encoding — 字符集编码
  将字符集中整数转换到其他形式以在计算机中表示
  对字符集中整数值进行编码的目的是为了更好的保存和传输字符
- Language
  定义字符集中字符的使用规则
  例如：字符对应的大写或小写形式，字符的顺序等

关于字符集的三个基本的概念

- Character Set — 字符集
  字符与整数之间的映射
  如果字符集的定义中每一个字符使用超过 8 bit 的整数，则该字符集被称为 Multibyte Character Set
- Character set encoding — 字符集编码
  将字符集中整数转换到其他形式以在计算机中表示
  对字符集中整数值进行编码的目的是为了更好的保存和传输字符
- Language
  定义字符集中字符的使用规则
  例如：字符对应的大写或小写形式，字符的顺序等

为什么国际化如此困难

- 计算机发展的初期没有考虑过这个问题，仅仅提供了英语的支持，一个字节最多只能表示 256 个字符
- 全世界许多的书写系统，而不同书写系统规则差异很大
  - 字母 vs. 表意文字
  - 从左到右 vs. 从右到左
  - 是否区分大小写，是否使用空格分隔……
- Unicode 出现之前为了能够支持不同的语言，各种语言使用不同的字符集和字符集编码，甚至同一种语言都存在多种字符集和字符集编码（仅中文就有 GBK、BIG5 等）要支持多种语言，就必须同时处理多种字符集编码

为什么国际化如此困难

- 计算机发展的初期没有考虑过这个问题，仅仅提供了英语的支持，一个字节最多只能表示 256 个字符
- 全世界许多的书写系统，而不同书写系统规则差异很大
  - 字母 vs. 表意文字
  - 从左到右 vs. 从右到左
  - 是否区分大小写，是否使用空格分隔……
- Unicode 出现之前为了能够支持不同的语言，各种语言使用不同的字符集和字符集编码，甚至同一种语言都存在多种字符集和字符集编码（仅中文就有 GBK、BIG5 等）要支持多种语言，就必须同时处理多种字符集编码

为什么国际化如此困难

- 计算机发展的初期没有考虑过这个问题，仅仅提供了英语的支持，一个字节最多只能表示 256 个字符
- 全世界许多的书写系统，而不同书写系统规则差异很大
  - 字母 vs. 表意文字
  - 从左到右 vs. 从右到左
  - 是否区分大小写，是否使用空格分隔.....
- Unicode 出现之前为了能够支持不同的语言，各种语言使用不同的字符集和字符集编码，甚至同一种语言都存在多种字符集和字符集编码（仅中文就有 GBK、BIG5 等）要支持多种语言，就必须同时处理多种字符集编码

Outline

Outline

MAKE YOUR PROGRAM LOCALE AWARE

- 通过调用 `setlocale()` 设置 locale，以使 C 库函数能够 locale aware. 如果不调用则使用默认的 locale – C
- locale 信息决定了许多 C 库函数的行为
  eg. `strftime()` (`ctime()` is not local aware!)
  `strfmon()` (格式化钱币数值)
- locale 分为许多类别，每个类别分别决定不同的函数的行为
  eg. `strftime()` – LC_TIME
  `strcoll()` – LC_COLLATE

MAKE YOUR PROGRAM LOCALE AWARE

- 通过调用 `setlocale()` 设置 locale，以使 C 库函数能够 locale aware. 如果不调用则使用默认的 locale – C
- locale 信息决定了许多 C 库函数的行为
  eg. `strftime()` (`ctime()` is not local aware!)
  `strfmon()` (格式化钱币数值)
- locale 分为许多类别，每个类别分别决定不同的函数的行为
  eg. `strftime()` – LC_TIME
  `strcoll()` – LC_COLLATE

MAKE YOUR PROGRAM LOCALE AWARE

- 通过调用 `setlocale()` 设置 locale，以使 C 库函数能够 locale aware. 如果不调用则使用默认的 locale – C
- locale 信息决定了许多 C 库函数的行为
  eg. `strftime()` (`ctime()` is not local aware!)
  `strfmon()` (格式化钱币数值)
- locale 分为许多类别，每个类别分别决定不同的函数的行为
  eg. `strftime()` – LC_TIME
  `strcoll()` – LC_COLLATE

C LIBRARY AND WCHAR_T

- C99 引入了 `wchar_t` 来处理多字节字符集，`wchar_t` 占多少个 bit 由具体实现而决定
  （gcc 中为 32-bit，且直接保存 Unicode 字符集中定义的整数值）
- `wchar.h` 中定义了一组类似于 `ctype.h` 中的函数用于处理 `wchar_t`
  eg. `wcslen()` - `strlen()`
  `wprintf()` - `printf()`
- 另外有一组用于多字节编码字节流和 `wchar_t` 字符串之间进行转换的函数，这些函数的行为也取决于 locale 的设置
  eg. `mbstowcs()` (convert `wchar_t`)
  `wcstombs` (convert to multi-byte string)

C LIBRARY AND WCHAR_T

- C99 引入了 `wchar_t` 来处理多字节字符集，`wchar_t` 占多少个 bit 由具体实现而决定
  （gcc 中为 32-bit，且直接保存 Unicode 字符集中定义的整数值）
- `wchar.h` 中定义了一组类似于 `ctype.h` 中的函数用于处理 `wchar_t`
  eg. `wcslen()` - `strlen()`
  `wprintf()` - `printf()`
- 另外有一组用于多字节编码字节流和 `wchar_t` 字符串之间进行转换的函数，这些函数的行为也取决于 locale 的设置
  eg. `mbstowcs()` (convert `wchar_t`)
  `wcstombs` (convert to multi-byte string)

C LIBRARY AND WCHAR_T

- C99 引入了 `wchar_t` 来处理多字节字符集，`wchar_t` 占多少个 bit 由具体实现而决定
  （gcc 中为 32-bit，且直接保存 Unicode 字符集中定义的整数值）
- `wchar.h` 中定义了一组类似于 `ctype.h` 中的函数用于处理 `wchar_t`
  eg. `wcslen()` - `strlen()`
  `wprintf()` - `printf()`
- 另外有一组用于多字节编码字节流和 `wchar_t` 字符串之间进行转换的函数，这些函数的行为也取决于 locale 的设置
  eg. `mbstowcs()` (convert `wchar_t`)
  `wcstombs` (convert to multi-byte string)

PRONS AND CONS ABOUT THIS APPROACH

- prons
  - Provides a portable way to handle different character sets and encodings.
  - The user can use different character sets and the programmer don't need to care about it.
  - Programmers don't need to write code to handle character sets and encodings directly.
- cons
  - The library function needs to handle so many character sets, it may not be very efficient.
  - `wchar_t` requires more memory.
  - Because the above reason, it should not be used to store text on disk or transport on the net.
  - Hard to include characters used in different languages in the same text file.

PRONS AND CONS ABOUT THIS APPROACH

- prons
  - Provides a portable way to handle different character sets and encodings.
  - The user can use different character sets and the programmer don't need to care about it.
  - Programmers don't need to write code to handle character sets and encodings directly.
- cons
  - The library function needs to handle so many character sets, it may not be very efficient.
  - `wchar_t` requires more memory.
  - Because the above reason, it should not be used to store text on disk or transport on the net.
  - Hard to include characters used in different languages in the same text file.

Outline

A LITTLE HISTORY ABOUT UNICODE

Unicode — unique, universal, and uniform character encoding
"*Begin at 0 and add the next character*"

- The concept of universal code is not new
  Xerox Star use 16-bit character encoding in 1981, and went on 27 languages including Chinese, Japanese.
- The need of Unicode begans very early
  The story of Mark Davis making up Apple KanjiTalk in 1985
- The initial work was done by 3 engineers
  Joe Becker(Xerox), Lee Collins(Xerox), Mark Davis(Apple).
  This was in 1987.

A LITTLE HISTORY ABOUT UNICODE

Unicode — unique, universal, and uniform character encoding
"*Begin at 0 and add the next character*"

- The concept of universal code is not new
  Xerox Star use 16-bit character encoding in 1981, and went on 27 languages including Chinese, Japanese.
- The need of Unicode begans very early
  The story of Mark Davis making up Apple KanjiTalk in 1985
- The initial work was done by 3 engineers
  Joe Becker(Xerox), Lee Collins(Xerox), Mark Davis(Apple).
  This was in 1987.

A LITTLE HISTORY ABOUT UNICODE

Unicode — unique, universal, and uniform character encoding
"*Begin at 0 and add the next character*"

- The concept of universal code is not new
  Xerox Star use 16-bit character encoding in 1981, and went on 27 languages including Chinese, Japanese.
- The need of Unicode begans very early
  The story of Mark Davis making up Apple KanjiTalk in 1985
- The initial work was done by 3 engineers
  Joe Becker(Xerox), Lee Collins(Xerox), Mark Davis(Apple). This was in 1987.

A LITTLE HISTORY ABOUT UNICODE

The latest version of Unicode is 5.0.0

- The beginning of Unicode: Unicode 88
- Unifying CJK
  In 1988, discussion for the criteria for Han unification began at the Research Libraryies Group at Palo Alto.
- In 1991, the ISO Working Group responsible for ISO/IEC 10646 and the Unicode Consortium decided to create one universal standard for coding multilingual text.
- Later, major OSes began to support Unicode and more and more programmers began to use Unicode in their programs.

A LITTLE HISTORY ABOUT UNICODE

The latest version of Unicode is 5.0.0

- The beginning of Unicode: Unicode 88
- Unifying CJK
  In 1988, discussion for the criteria for Han unification began at the Research Libraryies Group at Palo Alto.
- In 1991, the ISO Working Group responsible for ISO/IEC 10646 and the Unicode Consortium decided to create one universal standard for coding multilingual text.
- Later, major OSes began to support Unicode and more and more programmers began to use Unicode in their programs.

A LITTLE HISTORY ABOUT UNICODE

The latest version of Unicode is 5.0.0

- The beginning of Unicode: Unicode 88
- Unifying CJK
  In 1988, discussion for the criteria for Han unification began at the Research Libraryies Group at Palo Alto.
- In 1991, the ISO Working Group responsible for ISO/IEC 10646 and the Unicode Consortium decided to create one universal standard for coding multilingual text.
- Later, major OSes began to support Unicode and more and more programmers began to use Unicode in their programs.

A LITTLE HISTORY ABOUT UNICODE

The latest version of Unicode is 5.0.0

- The beginning of Unicode: Unicode 88
- Unifying CJK
  In 1988, discussion for the criteria for Han unification began at the Research Libraryies Group at Palo Alto.
- In 1991, the ISO Working Group responsible for ISO/IEC 10646 and the Unicode Consortium decided to create one universal standard for coding multilingual text.
- Later, major OSes began to support Unicode and more and more programmers began to use Unicode in their programs.

TECHNICAL DETAILS ON UNICODE

- Unicode vs. ISO10646
  - ISO10646 只定义了整数到字符的映射，定义了 2 种编码 UCS-2 和 UCS-4，该映射关系将和 Unicode 保持统一
  - Unicode 定义了其他更加易于使用的编码，提供了如何正确显示字符的规则和信息
- A significant advantage of Unicode
  Unicode's encoding doesn't use shift states, so a loss of data in the middle does not corrupt the subsequent encoded data.
- A common pitfall — Unicode is 16-bit?
  - It's true in the early days of Unicode
  - But Unicode has now defined more than 100,000 characters
  - Apparently, it can't fit into an 16-bit integer

TECHNICAL DETAILS ON UNICODE

- Unicode vs. ISO10646
  - ISO10646 只定义了整数到字符的映射，定义了 2 种编码 UCS-2 和 UCS-4，该映射关系将和 Unicode 保持统一
  - Unicode 定义了其他更加易于使用的编码，提供了如何正确显示字符的规则和信息
- A significant advantage of Unicode
  Unicode's encoding doesn't use shift states, so a loss of data in the middle does not corrupt the subsequent encoded data.
- A common pitfall — Unicode is 16-bit?
  - It's true in the early days of Unicode
  - But Unicode has now defined more than 100,000 characters
  - Apparently, it can't fit into an 16-bit integer

Technical details on Unicode

- Unicode vs. ISO10646
  - ISO10646 只定义了整数到字符的映射，定义了 2 种编码 UCS-2 和 UCS-4，该映射关系将和 Unicode 保持统一
  - Unicode 定义了其他更加易于使用的编码，提供了如何正确显示字符的规则和信息
- A significant advantage of Unicode
  Unicode's encoding doesn't use shift states, so a loss of data in the middle does not corrupt the subsequent encoded data.
- A common pitfall — Unicode is 16-bit?
  - It's true in the early days of Unicode
  - But Unicode has now defined more than 100,000 characters
  - Apparently, it can't fit into an 16-bit integer

The division Unicode

The whole Unicode character set is divided into 17 planes.

- Each plane has 2^{16} characters
- Basic Multilingual Plane (BMP)
  Plane 0 covers all the characters that can be used by programmers before Unicode. The coverage of character is done from west to east.
- Astral planes
  Plane 1 through 16 contains characters that are not offen used. eg. Musical symbol, acient Greek
- It's unlikely you will need to handle characters outside BMP in the near term, but its unwise to make such assumption.

THE DIVISION UNICODE

The whole Unicode character set is divided into 17 planes.

- Each plane has 2^{16} characters
- Basic Multilingual Plane (BMP)
  Plane 0 covers all the characters that can be used by programmers before Unicode. The coverage of character is done from west to east.
- Astral planes
  Plane 1 through 16 contains characters that are not offen used. eg. Musical symbol, acient Greek
- It's unlikely you will need to handle characters outside BMP in the near term, but its unwise to make such assumption.

The division Unicode

The whole Unicode character set is divided into 17 planes.

- Each plane has 2^{16} characters
- Basic Multilingual Plane (BMP)
  Plane 0 covers all the characters that can be used by programmers before Unicode. The coverage of character is done from west to east.
- Astral planes
  Plane 1 through 16 contains characters that are not offen used. eg. Musical symbol, acient Greek
- It's unlikely you will need to handle characters outside BMP in the near term, but its unwise to make such assumption.

THE DIVISION UNICODE

The whole Unicode character set is divided into 17 planes.

- Each plane has 2^{16} characters
- Basic Multilingual Plane (BMP)
  Plane 0 covers all the characters that can be used by programmers before Unicode. The coverage of character is done from west to east.
- Astral planes
  Plane 1 through 16 contains characters that are not offen used. eg. Musical symbol, acient Greek
- It's unlikely you will need to handle characters outside BMP in the near term, but its unwise to make such assumption.

About canonical form

- A character in Unicode may have more than one representation.
- There are such kind of things called *combining character sequence* which is a base character followed by any number of combining characters.
- Different combination can represent the same thing, only one form of them is called canonical form. (For more details, refer to Unicode FAQ)

About canonical form

- A character in Unicode may have more than one representation.
- There are such kind of things called *combining character sequence* which is a base character followed by any number of combining characters.
- Different combination can represent the same thing, only one form of them is called canonical form. (For more details, refer to Unicode FAQ)

About canonical form

- A character in Unicode may have more than one representation.
- There are such kind of things called *combining character sequence* which is a base character followed by any number of combining characters.
- Different combination can represent the same thing, only one form of them is called canonical form. (For more details, refer to Unicode FAQ)

Various encodings for Unicode

UTF stands for *UCS Transformation Format*, where UCS stands for *Universal (Multi-Octet Coded) Character Set*.

- UTF-1 (Not used), UTF-7 (For SMTP)
- UTF-8 (defined in RFC2279)
- UTF-16 (vs. UCS-2, defined in RFC2781)
- UTF-32 (vs. UCS-4)
- UTF-9, UTF-18 (Defined in RFC4042, they are new)

The most widely used encodings now are UTF-8 and UTF-16.

Overview of UTF-16 encoding

UTF-16 encoding is aimed to encode all the characters in BMP in exacly two octets, and encode all the other characters in the next 16 planes in exactly four octets.

The encoding rules:

- Characters with values less than 0x10000 (Those in BMP) are represented as a 16-bit integer with a value the same with the character number. (less than 2^{16} characters can be encoded)

Overview of UTF-16 encoding

UTF-16 encoding is aimed to encode all the characters in BMP in exacly two octets, and encode all the other characters in the next 16 planes in exactly four octets.
The encoding rules:

- Characters with values less than 0x10000 (Those in BMP) are represented as a 16-bit integer with a value the same with the character number. (less than 2^{16} characters can be encoded)

UTF-16 encoding Rules cont'

- Characters with values between 0x10000 and 0x10FFFF are represented with two 16-bit integers.
  - High-half one's value is between 0xD800 and 0xDBFF.
  - Low-half one's between 0xDC00 and 0xDFFF.

  These two blocks are reserved in BMP for use by UTF-16 and are called high and low surrogate area respectively. (2^{20} characters can be encoded)
- Characters with values greater than 0x10FFFF cannot be encoded in UTF-16. (So UTF-16 can only encode a total of $17 \times 2^{16} - 2^{11}$ characters.)

UTF-16 encoding Rules cont'

- Characters with values between 0x10000 and 0x10FFFF are represented with two 16-bit integers.
  - High-half one's value is between 0xD800 and 0xDBFF.
  - Low-half one's between 0xDC00 and 0xDFFF.

  These two blocks are reserved in BMP for use by UTF-16 and are called high and low surrogate area respectively. (2^{20} characters can be encoded)
- Characters with values greater than 0x10FFFF cannot be encoded in UTF-16. (So UTF-16 can only encode a total of $17 \times 2^{16} - 2^{11}$ characters.)

THE ENCODING PROCESS

Encoding characters with value less than 0x10000 is trivial. For characters larger than 0x10000, denote one as U, the encoding process is as follows

1. $U' = U - 0x10000 = \mathtt{yyyyyyyyyy}\ \mathtt{xxxxxxxxxx}$
2. $W1 = 110110\ \mathtt{0000000000}$
   $W2 = 110111\ \mathtt{0000000000}$
3. $W1 = 110110\ \mathtt{yyyyyyyyyy}$
   $W2 = 110111\ \mathtt{xxxxxxxxxx}$

THE ENCODING PROCESS

Encoding characters with value less than 0x10000 is trivial. For characters larger than 0x10000, denote one as U, the encoding process is as follows

1. $U' = U - 0x10000 = \texttt{yyyyyyyyyy}\ \ \texttt{xxxxxxxxxx}$
2. $W1 = 110110\ \texttt{0000000000}$
   $W2 = 110111\ \texttt{0000000000}$
3. $W1 = 110110\ \texttt{yyyyyyyyyy}$
   $W2 = 110111\ \texttt{xxxxxxxxxx}$

THE ENCODING PROCESS

Encoding characters with value less than 0x10000 is trivial. For characters larger than 0x10000, denote one as U, the encoding process is as follows

1. $U' = U - 0x10000 = \mathtt{yyyyyyyyyy}\ \mathtt{xxxxxxxxxx}$
2. $W1 = 110110\ \mathtt{0000000000}$
   $W2 = 110111\ \mathtt{0000000000}$
3. $W1 = 110110\ \mathtt{yyyyyyyyyy}$
   $W2 = 110111\ \mathtt{xxxxxxxxxx}$

THE ENCODING PROCESS

Encoding characters with value less than 0x10000 is trivial. For characters larger than 0x10000, denote one as U, the encoding process is as follows

1. $U' = U - 0x10000 = \texttt{yyyyyyyyyy}\ \ \texttt{xxxxxxxxxx}$
2. $W1 = 110110\ \texttt{0000000000}$
   $W2 = 110111\ \texttt{0000000000}$
3. $W1 = 110110\ \texttt{yyyyyyyyyy}$
   $W2 = 110111\ \texttt{xxxxxxxxxx}$

Big-endian or Little-endian?

- Since UTF-16 uses 2 bytes as an encoding unit, the endian problem must be dealt with.
- 0xFFFE can not appear in Unicode while 0xFEFF represents "ZERO WIDTH NON-BREAKING SPACE". It is prepended to a Unicode character stream as the Byte Order Marker (BOM).
  - If you encount a two bytes sequence 0xFE 0xFF, the the encoding is in big-endian.
  - If you encount 0xFF 0xFE, then the encoding is in little-endian.
- UTF-16BE and UTF-16LE are defined to label the UTF-16 encoded files whether they are encoded in big-endian or little-endian.

Big-endian or Little-endian?

- Since UTF-16 uses 2 bytes as an encoding unit, the endian problem must be dealt with.
- 0xFFFE can not appear in Unicode while 0xFEFF represents "ZERO WIDTH NON-BREAKING SPACE". It is prepended to a Unicode character stream as the Byte Order Marker (BOM).
  - If you encount a two bytes sequence 0xFE 0xFF, the the encoding is in big-endian.
  - If you encount 0xFF 0xFE, then the encoding is in little-endian.
- UTF-16BE and UTF-16LE are defined to label the UTF-16 encoded files whether they are encoded in big-endian or little-endian.

Big-endian or Little-endian?

- Since UTF-16 uses 2 bytes as an encoding unit, the endian problem must be dealt with.
- 0xFFFE can not appear in Unicode while 0xFEFF represents "ZERO WIDTH NON-BREAKING SPACE". It is prepended to a Unicode character stream as the Byte Order Marker (BOM).
  - If you encount a two bytes sequence 0xFE 0xFF, the the encoding is in big-endian.
  - If you encount 0xFF 0xFE, then the encoding is in little-endian.
- UTF-16BE and UTF-16LE are defined to label the UTF-16 encoded files whether they are encoded in big-endian or little-endian.

Overview of UTF-8 encoding

- UTF-8 was originally a project aimed to specify a File System Safe UCS Transformation Format that is compatible UNIX system. (Plan 9 is the first one to use UTF-8 though it's not UNIX.)
- UTF-8 encodes UCS-2 or UCS-4 characters as a varying number of octets. The number of octets range from 1 to 6.
- Characteristics
  - UTF-8 encoded characters are just bytes stream.
  - 7-bit US-ASCII characters has the same value defined in Unicode as they were in the ASCII table.
  - US-ASCII values do not appear otherwise in UTF-8 encoded character stream.

Overview of UTF-8 encoding

- UTF-8 was originally a project aimed to specify a File System Safe UCS Transformation Format that is compatible UNIX system. (Plan 9 is the first one to use UTF-8 though it's not UNIX.)
- UTF-8 encodes UCS-2 or UCS-4 characters as a varying number of octets. The number of octets range from 1 to 6.
- Characteristics
  - UTF-8 encoded characters are just bytes stream.
  - 7-bit US-ASCII characters has the same value defined in Unicode as they were in the ASCII table.
  - US-ASCII values do not appear otherwise in UTF-8 encoded character stream.

Overview of UTF-8 encoding

- UTF-8 was originally a project aimed to specify a File System Safe UCS Transformation Format that is compatible UNIX system. (Plan 9 is the first one to use UTF-8 though it's not UNIX.)
- UTF-8 encodes UCS-2 or UCS-4 characters as a varying number of octets. The number of octets range from 1 to 6.
- Characteristics
  - UTF-8 encoded characters are just bytes stream.
  - 7-bit US-ASCII characters has the same value defined in Unicode as they were in the ASCII table.
  - US-ASCII values do not appear otherwise in UTF-8 encoded character stream.

How UTF-8 encoding is done

Besides ASCII values, the first byte indicates how many bytes the character takes, the left bytes start with 10.

UCS-4 range (hex.)	UTF-8 octet sequence (bin.)
0000 0000–0000 007F	0xxxxxxx
0000 0080–0000 07FF	110xxxxx 10xxxxxx
0000 0800–0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000–001F FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0020 0000–03FF FFFF	111110xx .....
0400 0000–7FFF FFFF	1111110x .....

Some notes on UTF-8 encoding

- When converting from UTF-16 to UTF-8 , values between D800 – DFFF should be special treated. The UTF-16 encoding should be first undone.
- FF and FE will not appear in UTF-8 encoding.
- Security Considerations. Must handle invalid byte stream. IIS once has secure problem with UTF-8 encoded byte streams.

Choose an internal encoding – UTF-16

UTF-16

- pros
  - Java, C# use UTF-16 as it's internal encoding.
  - Effecient dealing with Asian characters.
  - When you are writing new application and is using Java or C#, you should consider use UTF-16.
- cons
  - Uncompatible with already existing C libraries.
  - When converting old programs to use Unicode, lots of change are needed.

Choose an internal encoding – UTF-16

UTF-16

- pros
  - Java, C# use UTF-16 as it's internal encoding.
  - Effecient dealing with Asian characters.
  - When you are writing new application and is using Java or C#, you should consider use UTF-16.
- cons
  - Uncompatible with already existing C libraries.
  - When converting old programs to use Unicode, lots of change are needed.

CHOOSE AN INTERNAL ENCODING – UTF-8

UTF-8

- pros
  - C programs appreciate UTF-8 since there are no portable way to define an 16-bit integer in C.
  - Compatible with UNIX system's most library.
  - When you are modifying old programs to support unicode, using UTF-8 only need small changes. Some even need no change. (eg. `cat, echo`)
- cons
  - When dealing lots of Asian character, it's not so effecient.
  - To locate a specific character in the bytes stream, you need to iterate from the start of the stream. (UTF-16 also have the problem, less serious.)

Choose an internal encoding – UTF-8

UTF-8

- pros
  - C programs appreciate UTF-8 since there are no portable way to define an 16-bit integer in C.
  - Compatible with UNIX system's most library.
  - When you are modifying old programs to support unicode, using UTF-8 only need small changes. Some even need no change. (eg. `cat, echo`)
- cons
  - When dealing lots of Asian character, it's not so effecient.
  - To locate a specific character in the bytes stream, you need to iterate from the start of the stream. (UTF-16 also have the problem, less serious.)

Other consideration

- What encoding does your library support
  eg. The ncurses library.
- Convert external encoding into internal encoding
  - User's input, files etc. may not be encoded as the same with your internal encoding.
  - `iconv()` can convert the byte streams between various encodings.
- User input is not one byte a time now! It's multi-byte a time.
- Do you need to treat wide characters specially?
  eg. In ncurses library, wide characters take two cursor positions.

Other consideration

- What encoding does your library support
  eg. The ncurses library.
- Convert external encoding into internal encoding
  - User's input, files etc. may not be encoded as the same with your internal encoding.
  - `iconv()` can convert the byte streams between various encodings.
- User input is not one byte a time now! It's multi-byte a time.
- Do you need to treat wide characters specially?
  eg. In ncurses library, wide characters take two cursor positions.

Other consideration

- What encoding does your library support
  eg. The ncurses library.
- Convert external encoding into internal encoding
  - User's input, files etc. may not be encoded as the same with your internal encoding.
  - `iconv()` can convert the byte streams between various encodings.
- User input is not one byte a time now! It's multi-byte a time.
- Do you need to treat wide characters specially?
  eg. In ncurses library, wide characters take two cursor positions.

Other consideration

- What encoding does your library support
  eg. The ncurses library.
- Convert external encoding into internal encoding
  - User's input, files etc. may not be encoded as the same with your internal encoding.
  - `iconv()` can convert the byte streams between various encodings.
- User input is not one byte a time now! It's multi-byte a time.
- Do you need to treat wide characters specially?
  eg. In ncurses library, wide characters take two cursor positions.

Some Libraries that can be used to handle Unicode

- Glib
  - One of the fundamental libraries for the GNOME project, provides various routines to deal with UTF-8, UTF-16 encoded characters.
  - Very handy if you are using C since it also provides many useful data structure and other things.
- ICU — International Components for Unicode
  - From IBM, now open sourced.
  - Supports C/C++, Java.
  - Powerful and feature rich. Has support for Unicode bi-direction algorithm.

Some Libraries that can be used to handle Unicode

- Glib
  - One of the fundamental libraries for the GNOME project, provides various routines to deal with UTF-8, UTF-16 encoded characters.
  - Very handy if you are using C since it also provides many useful data structure and other things.
- ICU — International Components for Unicode
  - From IBM, now open sourced.
  - Supports C/C++, Java.
  - Powerful and feature rich. Has support for Unicode bi-direction algorithm.

Outline

Outline

Introduction to gettext

Making your program support Unicode is just the i18n part. The second part is I10n, provides locale specific infomation for the program.

- POSIX standardized *catgets*, but it's hard to use.
- *gettext* is the de facto standard in the GNU world, based on a design originally done by Sun for Solaris.
- *gettext* can translate program message into different languages at runtime.

TRASLATION MECHANISM AND FUNCTIONS

The translation mechanism is to use the string that appears in the source code as the key, look up the translated message from some file.

- `textdomain()` is used to pick up the file which contains the translated message for the application.
- `gettext()` is responsible for looking up the the translated message by the key. It's used against literature strings in the source code.

TRASLATION MECHANISM AND FUNCTIONS

The translation mechanism is to use the string that appears in the source code as the key, look up the translated message from some file.

- `textdomain()` is used to pick up the file which contains the translated message for the application.
- `gettext()` is responsible for looking up the the translated message by the key. It's used against literature strings in the source code.

OTHER FUNCTIONS AND TOOLS

- `bindtextdomain()` is used to specify the directory in which may contain the message file. The default directory is in /usr/share/locale. Use this when you are testing and.
- `xgettext` is used to extract all the string need to be translated.
- `msgfmt` is used to compile the translated message into binary file that can be used by `gettext()`.

OTHER FUNCTIONS AND TOOLS

- `bindtextdomain()` is used to specify the directory in which may contain the message file. The default directory is in /usr/share/locale. Use this when you are testing and.
- `xgettext` is used to extract all the string need to be translated.
- `msgfmt` is used to compile the translated message into binary file that can be used by `gettext()`.

Outline

STEPS USING GETTEXT

1. Adopt the gettext.h header file into your application. Add the following definition to a header file included by all your C source files.

```
#define ENABLE_NLS 1
#include "gettext.h"
#define _(msgid) gettext(msgid)
#define N_(msgid) msgid
```

The last two macros are just convention, but they also make your life easier when you want to do the translation.

STEPS USING GETTEXT CONT'

2. Call `setlocale()` as appropriate. The easiest is call `setlocale(LC_ALL, "")`.
3. Pick a text domain for the application and set it with `textdomain()`.
4. If testing, call `bindtextdomain()` and bind the text domain to a particular directory.
5. Use `strfmon()`, `strftime()` and the `'` flag for `printf()` as appropriate.

STEPS USING GETTEXT CONT'

2. Call `setlocale()` as appropriate. The easiest is call `setlocale(LC_ALL, "")`.
3. Pick a text domain for the application and set it with `textdomain()`.
4. If testing, call `bindtextdomain()` and bind the text domain to a particular directory.
5. Use `strfmon()`, `strftime()` and the `'` flag for `printf()` as appropriate.

STEPS USING GETTEXT CONT'

2. Call `setlocale()` as appropriate. The easiest is call `setlocale(LC_ALL, "")`.
3. Pick a text domain for the application and set it with `textdomain()`.
4. If testing, call `bindtextdomain()` and bind the text domain to a particular directory.
5. Use `strfmon()`, `strftime()` and the `'` flag for `printf()` as appropriate.

STEPS USING GETTEXT CONT'

2. Call `setlocale()` as appropriate. The easiest is call `setlocale(LC_ALL, "")`.
3. Pick a text domain for the application and set it with `textdomain()`.
4. If testing, call `bindtextdomain()` and bind the text domain to a particular directory.
5. Use `strfmon()`, `strftime()` and the `'` flag for `printf()` as appropriate.

STEPS USING GETTEXT CONT'

6. Mark all static literature strings use `N_()`, other literature strings use `_()` (Of course, you only need to mark the strings that need to be translated.)
7. Use `xgettext` to extract strings needs to be translated.

```
$ xgettext -k=_ -k=N_ \
> -default-domain=domainname *.c
```

You will get a file `domainname.po`.

STEPS USING GETTEXT CONT'

6. Mark all static literature strings use `N_()`, other literature strings use `_()` (Of course, you only need to mark the strings that need to be translated.)
7. Use `xgettext` to extract strings needs to be translated.

```
$ xgettext -k=_ -k=N_ \
> -default-domain=domainname *.c
```

You will get a file `domainname.po`.

STEPS USING GETTEXT CONT'

8. Make a copy of `domainname.po`, say `foo.po` and translate all the messages in that file.
9. Use `msgfmt` to compile `foo.po`.

```
$ msgfmt foo.po -o domainname.mo
```

Then you can put it to the right place and test it or use it. eg. Put it to /usr/share/locale/zh_CN/LC_MESSAGES if your translation is for Chinese. If testing, put it at ./zh_CN/LC_MESSAGES.

STEPS USING GETTEXT CONT'

8. Make a copy of `domainname.po`, say `foo.po` and translate all the messages in that file.
9. Use `msgfmt` to compile `foo.po`.
   ```
   $ msgfmt foo.po -o domainname.mo
   ```
   Then you can put it to the right place and test it or use it. eg. Put it to /usr/share/locale/zh_CN/LC_MESSAGES if your translation is for Chinese. If testing, put it at ./zh_CN/LC_MESSAGES.

Outline

Other libraries or tools for l10n

Each major user interface toolkit has itw own way to solve the problem of I10n. gettext is for command line interface.

- Qt
  Mark all literature string using `tr()` and use *Linguist* to do the translation.
- Pango
  It's the core of text and font handling for GTK+-2.x.
- ICU
  It also provides methods for I10n.

References and online resources


Arnold Robbins, *Linux Programming by Example*.


Unicode, Inc. *Ten years of Unicode 1988 - 1998*.


Tim Bray, *On the goodness of Unicode*.


Tim Bray, *Characters vs. Bytes*.


IETF RFC2279, *UTF-8, a transformation format of ISO 10646*


IETF RFC2781, *UTF-16, an encoding of ISO 10646*

i18n your program from now on

Thanks for your time