Про

и почему проблем не стало меньше

Почти реферат

Торгашов С. (stas.torgashov@outlook.com)

640 килобайт хватит всем или 16 бит хватит всем

В 1988 году Джозеф Бекер написал основополагающую работу, с которой, пожалуй, можно отсчитывать историю стандарта Unicode. Вероятно, этой работой мы в итоге обязаны тому, что 16-битная кодировка была выбрана в WinAPI (Windows NT 3.1 – 1993 год), в Java (1995 год) и, впоследствии, 16-битный формат вошёл и в C# например.

Выбранное для этого стандарта название Unicode призвано символизировать единый, универсальный и однозначный способ кодирования символьной информации. Также иногда последовательность символов Unicode именуют Unitext. Забегая вперёд можно сказать, что это правильно: чтобы подчеркнуть отличие от привычных нам строк символов в привычных языках программирования.
Что в указанной работе Бекера ещё важно: было предложено разделить собственно символ – графему – от его начертания – глифа (или набора глифов, поскольку в общем случае символ может состоять из нескольких элементов). Можно сказать, что кодирование Unicode – это способ задать уникальный номер (код) для любого начертания, используемого для письма.

Предложенный Бекером способ кодирования нашёл отражение в формате UCS – Universal

Coded Character Set, способ кодирования с фиксированной длиной кода (fixed-length encoding). UCS – это первая редакция стандарта до версии 1.1 включительно.

Тогда казалось, что достаточно сделать переход от 8 бит к 16 битам и это решит проблемы человечества с кодировками используемых языков общения.

Тем не менее, науке с тех пор известен не только UCS-2, но и UCS-4 – 32-х битный вариант. Последний пришлось придумать, когда в информационную сферу массово пришли азиаты с их объёмными азбуками и 16 бит категорически перестало хватать.
Здесь сразу отметим очевидные преимущества UCS-2 и UCS-4 – постоянная длина кодов и простые алгоритмы кодирования, а потому высокая скорость обработки текста в таких кодировках. Это же преимущество изрядно компенсировалось расточительностью в использовании памяти, особенно для европейских и близких к европейским азбук. Это очевидно для UCS-4, и верно (хотя и не столь очевидно) для UCS-2.

Итак, оказалось, что 16-бит недостаточно, а 32 бита *обычно* слишком много. И уже где-то примерно в 1996 году произошло изобретение UTF-16, который, в отличие от UCS-2, был в состоянии работать с кодами большей длины, чем 16-бит. Произошёл плавный и незаметный переход от кодировок с фиксированной длиной к кодировкам переменной длины, то есть, к UTF-16.
Именно UTF-16 используется для представления «широких» символов в Windows, равно как символьный тип в C# и Java, и в некоторых юникодных или использующих Unicode C++ библиотеках (Qt, ICU). О родных для C/C++ так называемых «широких» символах `wchar_t` и строках `std::wstring` надо будет поговорить отдельно.

В настоящее время стандарт Unicode определяет 109449 символов, причём около 74500 символов входят в так называемую группу CJK (China-Japan-Korea).

С момента появления UTF-16 начинается эпоха Unicode 2.0. UTF – это Unicode Transformation Format, формат кодирования с переменной шириной (variable-width encoding).

Небольшое необязательное отступление

ISO/IEC-10646 и Unicode – это не одно и то же. В целом, коды символов и способы кодирования совпадают в этих стандартах (более того, каждая редакция Unicode содержит ссылку на соответствующий ей 10646 документ), но Unicode дополнительно определяет ограничения на реализацию стандарта на разных платформах, алгоритмы, описания функциональных символов, которые не входят в ISO-10646.
И ещё. Тем кто застал ASCII и MS-DOS будет небезынтересно узнать, что ISO-8859-1 входит в ISO-10646 в качестве первых 256 пойнт-кодов.

Основные определения

Code point или пойнт-код (википедия, например, использует такой русскоязычный термин при описании ОКС-7; эта же калька выглядит пригодной и в данном случае) – это уникальный номер, представляющий любой символ или начертание, который мы используем или могли бы использовать при письме. Кодовый элемент (code unit) – это базовый элемент закодированной последовательности, который представляет из себя минимальную

комбинацию бит, кодирующую текстовый элемент, своего рода дескриптор символа. Звучит сложно, но для UTF-8 кодовый элемент состоит из 8 бит, для UTF-16 – из 16 бит, и для UTF-32 – из 32-х бит.
Абстрактный символ (abstract character) – это единица информации, которая используется для организации, представления или управления текстовыми данными.
Закодированный символ (encoded character) – это сопоставление абстрактного символа и code point. Один и тот же внешне одинаковый символ технически может иметь в Unicode разные кодировки или способы кодирования, например, греческая буква Ω (омега) или Ом — символ сопротивления в радиотехнике (например, 5Ω). И наоборот, один и тот же символ может иметь разные способы кодирования: используемое в испанском языке сочетание **l·** может кодироваться как один пойнт-код `U+0140` или два пойнт-кода `U+006C U+00B7` (причем второй вариант стандарт определяет как предпочтительный).
Суррогатная пара (surrogate code points) – это последовательность из двух 16-битных слов в кодировке UTF-16 для представления символов с кодами больше чем `U+FFFF` (диапазон допустимых значений юникода равен от `U+0000` до `U+10FFFF`), когда первым словом идёт код в диапазоне от `0xD800` до `0xDBFF`, а вторым — в диапазоне от `0xDC00` до `0xDFFF`. Таким образом, первая часть суррогатной пары сама по себе не имеет никакого смысла, она является корректным словом в UTF-16 только если за ней идёт вторая часть (отсюда и название «суррогатная пара»).
Пользовательский символ (user-perceived character) — это всё, что конечный пользователь воспринимает или готов воспринимать как отдельный письменный символ.
Пример: **ю́** – `U+044E` кириллическая строчная буква **ю**, за которой следует комбинирующий код ударения `U+0301`, то есть, собственно символ как его видит пользователь состоит из двух пойтн-кодов.
Кластер графем (grapheme cluster) – это последовательность закодированных символов, которые следует воспринимать вместе (например, **ch**).

Когда пользователь интересуется длиной текстовой строки его, как правило, интересует длина в пользовательских символах (user-perceived characters). Это настолько важный момент, что в некоторых API способ вычисления длины текста специфицируется отдельно (например, Twitter API содержит специальное упоминание о том, как именно высчитывается длина сообщения).

Ещё одно небольшое отступление

Важный момент стандарта Unicode, но не вполне очевидный с первого взгляда: отказ от рассмотрения текста как простого массива символов, то есть, привычных нам строк. Правильнее верить, что мы работаем с массивом глифов, а не графем. Такой подход немножко ломает стереотипы того, как надо программировать обработку и преобразование текста (определение длины строки, поиск подстроки в строке, разбиение, вставка символов и подстрок и прочее). Конечно, с точки зрения производительности кода (например, для подсчёта количества символов) правильнее использовать коды с фиксированной длиной или пошире (UCS-2, UTF-16, UTF-32), чтобы на хватило на всю или, по крайней мере, на заведомо большую часть предметной области, и кодировать сразу целые символы (графемы). Однако, Unicode определяет, что не все пойнт-коды соответствуют закодированным символам, некоторые могут быть *не*-символами (символы перемещения курсора, служебная информация и прочее). То есть, однозначно подсчитать не получится, можно *надеятся*, что в подавляющем большинстве случаев (а лучше даже *гарантировать* покрытие всей предметной области) результат будет корректным. Это же относится к операциям поиска

подстроки, разбиения, вставки и прочим.

Пример (на Java)

В немецком языке есть буква ß (эсцет). Её представление в Unicode – `U+00df`. Давайте посмотрим, чем будет равна длина строки, состоящей из одного этого символа:

```
System.out.println(“\u00df - ” + “\u00df”.length());
```

Результат вполне ожидаем — единица (кстати, мы помним, что в Java символы представлены в UTF-16). Теперь давайте приведём эсцет к верхнему регистру и снова возьмём длину строки:

```
System.out.println(“\u00df”.toUpperCase().length());
```
Получаем длину строки 2. Разгадка фокуса:

```
System.out.println(“\u00df”.toUpperCase() +
      “-” + “\u00df”.toUpperCase().length());
```

То есть, ß, приведённая к верхнему регистру, трансформировалась в SS. Теперь попробуем сделать обратное преобразование:

```
System.out.println(“\u00df”.toUpperCase().toLowerCase() +
      “-” + “\u00df”.toUpperCase().toLowerCase());
```

Вопреки казалось бы ожидаемому обратное преобразование не возвращает нам исходный символ ß, а оставляет две строчные буквы s.

Выводов из этой истории два: во-первых, даже казалось бы тривиальная операция `toupper` или `tolower` может изменить длину строки, а потому определить символ по его порядковому номеру в строке путём простой индексации без анализа содержимого в общем случае не получится (это иллюстрация к тезису, что текст это не просто массив символов); во-вторых, длина строки в Java считается как количество пойнт-кодов, а не символов. Последнее необходимо учитывать при обработке строк. А в случаях ручного управления памятью (например, в языках программирования C/C++ и подобных), придётся помнить, что необходимая длина буфера для хранения строки может измениться неочевидным образом в ходе преобразования текстовых строк.

BOM – Byte Order Mark

Эта штука появилась вследствие того, что некоторые варианты кодирования, а, точнее, все, кроме UTF-8, имеют размер кодового элемента больше одного байта. То есть, строго говоря, *только для* UTF-8 BOM, в принципе, не нужен. Нужно только джентльменское соглашение «вот мы тут везде используем UTF-8». Да и в нём — как будет показано ниже — нет особой необходимости, поскольку UTF-8 мало того, что легко детектируется, так он ещё и обладает свойством самосинхронизации. Об этом мы тоже скажем позднее.
А вот для UTF-16 и UTF-32 BOM иметь почти обязательно (можно, конечно, тоже как-то договориться, но на практике ошибок делается всё равно слишком много). Более того, по

понятным причинам для UTF-16 и UTF-32 нужно целых два BOM'а: для **L**ittle **E**ndian и для **B**ig **E**ndian; то есть, у нас должны быть определены маркеры UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE.

Итак, у нас в наличии следующий список BOM'ов:

UTF-8	0xEF 0xBB 0xBF
UTF-16BE	0xFE 0xFF
UTF-16LE	0xFF 0xFE
UTF-32BE	0x00 0x00 0xFE 0xFF
UTF-32LE	0xFF 0xFE 0x00 0x00

Интересная ситуация сложилась в Windows, которая, как мы теперь уже знаем, использует в своём API UTF-16 для представления юникодных символов и строк; они в WinAPI часто сопровождаются буквосочетанием WC(S) – Wide Character (String). При этом Windows не накладывает никаких ограничений на способы кодирования текстовой информации в прикладных программах, можно использовать как UTF-16BE, так и UTF-16LE, если выбрать и установить подходящий BOM. Но внутренний формат в Windows – всегда UTF-16LE, который используется в WinAPI для юникодных версий системных вызовов, он же используется для задания имн файлов в файловых системах NTFS и FAT (если используемый вариант последней поддерживает длинные имена файлов).

Кодировка UTF-8

К особенностям кодировки UTF-8 относят следующие свойства:

- Символ кодируется последовательностью байтов
- Байты последовательности, начиная со второго, всегда начинаются с битов `10`
- Если первый байт последовательности единственный, то он всегда начинается с бита `0`, то есть символ кодируется одним байтом и соответствует коду ASCII
- Если символ не ASCII, то первые биты первого байта содержат столько единиц, сколько байтов в последовательности (включая первый байт), после чего идёт бит `0`
- Все последующие значащие биты склеиваются в последовательность битов и интерпретируются как код символа

Таким образом, способ кодирования UTF-8 позволяет в потоке однозначно определить, что это а) Unicode, б) найти границы символов. Другие способы кодирования (UTF-16 и UTF-32) такими свойствами не обладают. К недостаткам UTF-8 частно относят необходимые расходы на перенос служебной информации (один-два старших бита в каждом байте).

Есть точка зрения, что эффективность и простота UTF-8 побудила Google определить `wchar_t` как тип длиной в 1 байт в Android NDK, чтобы в конце концов заставить пользователей отказаться от широких символов в пользу UTF-8. Авторитет Google сомнению не подлежит, но этот шаг всё равно не отменяет наличие `wchar_t` и проблем, с этим типом связанных. Потому что

Использование UTF-8 с `char` и `std::string`

С появлением UTF-16 и «широких» символов возникла необходимость добавить в C/C++ поддержку нового стандарта. По соображениям обратной совместимости и из-за равенства `sizeof(char) == 1` исправлять `char` не стали, а просто добавили новый тип строкового литерала `L`, тип `wchar_t` для «широких» символов и, соответственно, `std::wstring` для работы со строками таких символов. Практика показала, что решение вышло умеренно удачным.

Во-первых, `wchar_t` на разных платформах имеет разную длину. В Linux он, как правило, 32-х битный, в Windows размер `wchar_t` равен 16 битам, а в Android он и вовсе равен одному байту. Написание переносимого кода получило ещё одно препятствие. Плюс тип `wchar_t` (и `std::wstring`) имеет «вирусную» природу: неудобно использовать в одной программе одновременно и `char` (`std::string`) и `wchar_t` (`std::wstring`) из-за сложностей с взаимным перекодированием и использованием смешанных внешних API (частично с обычными символами, частично — с «широкими»).

В этом смысле UTF-8 представляет собой хорошую альтернативу новым типам символов и строк:

1. Минимальный размер кода символа — 1 байт, то есть, UTF-8 укладывается в единицы типов `char`, а это очень удобно
2. У нас остались привычные типы `char` и `std::string`
3. UTF-8 прозрачно совместим с ASCII
4. Корректная работа алгоритмов поиска подстроки в строке (не найдётся строка с началом в середине символа, например)
5. Кодировка в среднем достаточно компактна (в [3] приводятся сравнительные примеры размеров файлов в разных кодировках Unicode)
6. Там, где требуется использовать «широкие» символы или строки символов в API, можно выполнять перекодирование на лету; затраты на это обычно не слишком велики (рекомендации на эту тему есть в [3] и [5])

В недостатки использования UTF-8 можно записать необходимость перестраивать строку при

операциях вставки и удаления символа: использование UTF-8 может существенно замедлить выполнение кода, активно использующего такие операции. Но мы можем утешать себя тем, что — следует из общих рассуждений и примеров выше — преобразования текстовых строк в общем случае не тривиальны и, скорее всего, для каждого естественного языка придётся программировать такие операции специально.

В целом же, нет явных объективных причин предпочесть один определённый способ кодирования символов Unicode. По сумме факторов UTF-8 представляется в среднем наиболее удачным вариантом для использования. По крайней мере, его имеет смысл предпочесть в случаях, когда отсутствуют очевидные обоснованные или навязанные альтернативы.

Литература

1. http://unicode.org/
2. http://unicode.org/history/unicode88.pdf
3. http://utf8everywhere.org/
4. http://www.italiancpp.org/2016/04/20/unicode-localization-and-cpp-support/
5. https://habrahabr.ru/company/xakep/blog/257895/
6. https://habrahabr.ru/post/262679/