Gem #144: Немного об интерпретации серий байтов. Символы и схемы кодирования

Автор: Emmanuel Briot — AdaCore

Краткое содержание: В данном Gem описываются некоторые понятия, связанные с кодированием символов и Юникодом. В нем поясняется, почему существуют различные наборы символов, а также то, как работать с ними в случае необходимости обработки входных и исходящих данных на разных языках в приложении.

Давайте начнём…

Этот Gem начинается с разбора проблемы. Как французский уроженец, я часто манипулирую текстовыми файлами, которые содержат акцентированные буквы - буквы с диакритическим знаком (эти акценты, кстати, часто вводились как стенография для замены букв в словах, чтобы сохранить бумагу, когда она еще была дорогим товаром). К сожалению, в зависимости от того, как был создан файл, мои программы не обязательно видят одно и то же байтовое содержимое (которое зависит от кодировки и набора символов файла), и, если я просто попытаюсь отобразить их на экране (либо в текстовой консоли, либо в графическом окне), вывод может не выглядеть так, как я изначально ввел.

Глиф (Glyphs)

Глиф — это характерная особенность шрифта. В вычислительной технике — это элемент компьютерного символа, соответствующий графеме или графемоподобной единице текста: это может быть буква, число, знак пунктуации или пиктограмма, декоративный символ, графическая метка.

На этом этапе давайте представим понятие глиф. Это визуальные представления персонажей. Например, я хочу, чтобы "e-острый" (e-acute) выглядел как "E" с небольшим острым акцентом над ним. Это визуальное представление является конечной целью во многих приложениях, это то, что пользователь хочет видеть. В других приложениях, однако, глифы не имеют значения. Например, компилятору все равно, как символы отображаются на экране. Он должен знать, как разбить последовательность символов на слова и это все что ему надо. Предполагается, что консоль, в которой отображаются сообщения об ошибках, будет отображать те же символы, что и в исходном файле, если заданы те же байты, что и сам исходный файл.

Текстовый файл не встраивает описание того, как выглядит его представление. Вместо этого он состоит из байтов, которые объединяются определенным образом (иногда называемые схемами кодирования символов) для создания кодовых точек. Эти кодовые точки затем сопоставляются с определенным символом, используя набор символов какого-то национального языка. Наконец, шрифт определяет, как символ должен быть представлен в виде глифа.

Точное представление символа (его глиф) действительно зависит от используемого шрифта, поскольку "нижний регистр a" ("lower case a") может иметь совершенно разные аспекты, зависящие от шрифта. Но это выходит за рамки этой статьи.

В общем, ваше приложение не связано с отображением символов на символы с помощью шрифта. Об этом позаботится либо текстовая консоль, либо инструментарий GUI, который вы используете. Ваше приложение часто позволяет пользователю выбирать предпочтительный шрифт, а затем обязательно передавать допустимые символы. Инструментарий выполняет сложную работу по представлению символов. Например, эта работа является ролью инструментария `Pango` (доступного из `GtkAda`).

Наборы символов

Репертуар-это набор связанных символов, например алфавитов, используемых для написания английских или русских слов.

Набор символов - это отображение из репертуара в набор целых чисел, называемых кодовыми точками. Данный символ, как мы увидим, может существовать в нескольких различных наборах символов с разными кодовыми точками.

Большинство стандартных наборов символов (иногда сокращенно charsets) специфичны для одного языка. Например, существует стандарт ISO-8859-1 (также известный как латинский-1) и ISO-8859-15, которые используются для западноевропейских языков; мы также имеем ISO-8859-5 и с KOI8-R, которые имеют разные, но оба русские; окна внедрен ряд кодовых страниц, которые по сути являются наборами символов, специфичных для этой платформы; японские тексты часто используют ISO-2022-JP в, в то время как китайский имеет несколько стандартных наборов.

Возьмем самый простой из них - кодировку ASCII. Большинство разработчиков знакомы с ним. Например, в этом наборе кодовая точка 65 связана с буквой верхний регистр-A. Этот набор включает 128 символов, 31 из которых не имеют визуального представления. Он не содержит акцентированных букв, но в основном подходит для представления английских текстов.

Во многих западноевропейских языках, таких как французский, ASCII было недостаточно, поэтому ISO-8859-1 был построен поверх него. Первые 128 символов одинаковы, поэтому кодовая точка 65 по-прежнему имеет верхний регистр-A. Но она также добавляет 128 дополнительных символов, например 233- lower-case-e-with-acute. Смотрите страницу Википедии по ISO-8859-1 для более подробной информации.

Другим примером является ISO-8859-5 для русского текста, который несовместим с ISO-8859-1, хотя и основан на ASCII. Таким образом, 65-это все еще верхний регистр-A, но на этот раз 233-кириллица-маленькая буква-ща и lower-case-e-with-acute не существует.

В результате, если приложение читает кодированный файл ISO-8859-5, но считает, что это ISO-8859-1, оно отобразит недопустимый глиф для большинства русских букв, что, очевидно, сделает текст нечитаемым для пользователя.

В большинстве приложений (например, GPS IDE) есть способ указать, какой набор символов приложение должно ожидать по умолчанию, если файлы будут закодированы по умолчанию, и способ переопределить кодировку по умолчанию для определенных файлов.

Существует один набор символов, который включает все символы, существующие во всех других наборах символов (или, по крайней мере, предназначено для этого), и это Unicode (несколько сродни ISO-10646). Он включает в себя тысячи и тысячи символов (и все больше добавляются в каждой редакции Unicode), избегая при этом дубликатов. Для совместимости со многими существующими приложениями первые 256 символов такие же, как в ISO-8859-1, поэтому верхний регистр-a по-прежнему 65, а нижний регистр-e-with-acute по-прежнему 233. Но сейчас кириллица-мелкая буква-ща - 1097.

В настоящее время многие приложения (и даже языки программирования) будут систематически использовать Unicode внутри по умолчанию. Например, графический инструментарий GTK+ управляет Unicode только для внутренних строк, а также Python 3.X. Поэтому всякий раз, когда файл считывается с диска GPS, он сначала преобразуется из собственного набора символов в Unicode, а затем остальная часть приложения больше не должна заботиться о наборах символов.

Учитывая размер Unicode, существует несколько (если есть) шрифтов, которые могут представлять весь набор символов, но это не проблема в целом, так как большинство приложений не должны представлять египетские иероглифы...

Еще одной важной частью стандарта Unicode является набор таблиц для поиска свойств различных символов: какие из них следует рассматривать как пробел, как конвертировать из Нижнего в верхний регистр, какие буквы являются частью слов и т. д. Эти знания часто жестко закодированы в наших приложениях и часто включают серьезные изменения, когда приложение решает использовать Unicode внутри.

Схемы кодировки символов

Теперь мы знаем, как представлять символы как комбинацию кодовых точек и набора символов. Но нам часто нужно хранить эти символы в файлах, которые содержат только байты. Это кажется относительно легким, когда кодовая точка меньше 256, но становится гораздо менее очевидным для других кодовых точек, таких как 1097, которые мы видели ранее.

На практике этот вопрос решается несколькими способами. Схемы кодирования, такие как японский ISO-2022-JP, используют понятие сдвига плоскости: специальные байты указывают, что отныне байты должны интерпретироваться по-разному, до следующего сдвига плоскости. Поэтому для декодирования и кодирования требуется знание текущего состояния.

Сам Юникод позволяет определять три различные схемы кодирования (с их вариантами), которые известны как UTF-8, UTF-16 и UTF-32. Последнее число указывает количество битов, в которых кодируется каждый символ. Поэтому в UTF-32 каждый символ занимает четыре байта, что позволяет представить весь набор символов Юникода. Поэтому декодирование и кодирование тривиальны, но есть большая потеря пространства памяти на символ, связанная с UTF-32.

В UTF-16 каждый символ кодируется в два байта, что достаточно для всех символов, используемых разговорными языками. Другие символы предназначены для конкретного использования, например, египетские иероглифы. Для кодовых точек, которые не помещаются в два байта, Unicode определяет несколько специальных байтов (суррогатных пар), похожих на плоские сдвиги, описанные ранее. Таким образом, остается гораздо меньше свободного места, но декодирование и кодирование становится немного сложнее.

Вышеупомянутые две схемы кодирования не являются обратно совместимыми: приложение, которое было написано до Unicode и которое знало только о ASCII или ISO-8859-1, не поймет входные строки должным образом.

По этой причине и для экономии еще большего пространства памяти Unicode также определяет кодировку UTF-8. Для всех символов ASCII они по-прежнему представлены как до использования одного байта. Символы больше 127 кодируются как последовательность из нескольких байтов (и гарантируется, что все байты, кроме последнего, не являются частью ASCII).

Правильное управление строкой UTF-8 требует использования специализированных процедур (поскольку перемещение вперед одного символа означает перемещение вперед от 1 до 6 байтов). Однако случайное приложение может, например, перейти к следующему символу пробела, как это было раньше, перемещая вперед по одному байту за раз и останавливаясь, когда оно видит 32 (пробел) или 13 (новую строку). Это свойство часто может использоваться приложениями, которым не нужно представлять символы, как пример компилятора, о котором мы упоминали в начале.

Хотя понятия наборов символов и схем кодирования символов ортогональны, часто эти понятия объединяются. Например, когда кто-то упоминает ISO-8859-1, это обычно означает набор символов, а также его стандартное представление, где каждый символ представлен как один байт. Аналогично, кто-то говорит об UTF-8, как правило, означает набор символов Юникода вместе со схемой кодирования символов UTF-8.

Преобразования

Теперь у нас есть почти все части на месте, за исключением преобразования между наборами символов. Теоретически достаточно декодировать входной поток, используя правильную схему

кодирования символов, затем найти отображение для кодовых точек от начала до целевого набора символов и, наконец, использовать целевую схему кодирования для представления символов в виде байтов снова.

Когда символ не имеет отображения в целевой набор символов (например, E-acute в русском iso-8859-5), приложение должно решить, следует ли вызвать ошибку, игнорировать символ или найти транслитерацию (например, используя e' для e-acute).

Это, очевидно, утомительно и требует использования больших таблиц поиска для всех наборов символов, которые должно поддерживать ваше приложение.

В системах Unix существует стандартная библиотека `iconv` для выполнения этой работы преобразования от вашего имени. Проект GNU также предоставляет такую библиотеку с открытым исходным кодом для других систем.

Недавно мы добавили привязку к этой библиотеке в коллекцию компонентов GNAT (`GNATCOLL.Iconv`), что делает его еще проще в использовании для приложений созданных на языке Ada. Например:

```
with GNATCOLL.Iconv;   use GNATCOLL.Iconv;
procedure Main is
   EAcute : constant Character := Character'Val (16#E9#);
   --  in ISO-8859-1

   Result : constant String := Iconv
      ("Some string " & EAcute,
       To_Code   => UTF8,
       From_Code => ISO_8859_1);
begin
   null;
end Main;
```

XML/Ada также включал такие таблицы преобразования некоторое время, но поддерживает много меньше наборов символов. Проверьте `Unicode.CSS.*` пакеты.

Как Вы видите выше, мы снова используем в Ada строковый тип, последовательность не связана ни с каким определенным набором символов или схемой кодирования. В целом, как мы упомянули прежде, это не проблема, так как внутренне приложение будет использовать единственное кодирование (UTF-8 в большинстве случаев). Другой подход должен использовать `Wide_String` или `Wide_Wide_String`. К этому подходу применим тот же комментарий, что касается UTF-16 и UTF-32: они делают символьное управление более удобным, но за счет потраченной впустую памяти.

Управление строками UTF-8 и UTF-16

Последняя часть головоломки: как только у нас есть строка Unicode в памяти, - найти в ней каждый символ. Это требует специализированных подпрограмм, так как количество байтов является переменным для каждого символа.

Модуль Unicode XML/Ada включает в себя пакеты `Unicode.CES.*` это такой свой набор подпрограмм. В общем, идти по символам в строке вперед относительно легко и может быть сделано эффективно, в то время как идти назад в строке является более сложным и менее эффективным.

Библиотека времени выполнения (run-time library) GNAT также содержит такие пакеты, например `GNAT.Encode_UTF 8_String` и `GNAT.Decode_UTF8_String`. В частности, последний предоставляет `Decode_Wide_Character`, `Next_Wide_Character` и `Prev_Wide_Character`, чтобы найти все символы в строке.

Связанный со статьёй текст программы

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Автор: Emmanuel Briot — AdaCore

Эммануэль Брио работает в AdaCore с 1998 года. Он принимал участие в различных проектах, в частности, ориентированных на графические пользовательские интерфейсы, включая GtkAda, GPS, XML / Ada, GnatTracker и нашу внутреннюю CRM. Он получил степень инженера в Национальной школе телекоммуникаций - Брест, Франция (Ecole Nationale des Telecommunications - Brest, France).

Last Updated: 10/13/2017
Posted on: 3/25/2013

Обсуждение…