Декодер UTF-8 без ветвей

06 октября 2017 г.
nullprogram.com/blog/2017/10/06/

На этой неделе я попробовал написать декодер UTF-8 без ветвлений : функцию, которая декодирует одну кодовую точку UTF-8 из потока байтов без каких-либо `if`операторов, циклов, операторов короткого замыкания или других видов условных переходов. Вы можете найти исходный код здесь вместе с тестовым набором и бенчмарком:

- https://github.com/skeeto/branchless-utf8

В дополнение к декодированию следующей кодовой точки, он обнаруживает любые ошибки и возвращает указатель на следующую кодовую точку. Это полный пакет.

Почему безветвевой? Потому что высокопроизводительные ЦП конвейеризированы. То есть одна инструкция выполняется в несколько этапов, а многие инструкции выполняются в перекрывающихся временных интервалах, каждая на разном этапе.

Обычная аналогия — стирка. Вы можете иметь более одной загрузки белья в процессе одновременно, потому что стирка обычно является конвейерным процессом. Есть стадия стиральной машины, стадия сушилки и стадия складывания. Одна загрузка может быть в стиральной машине, вторая в сушилке, а третья складывается, и все это одновременно. Это значительно увеличивает пропускную способность, потому что в идеальных условиях с полным конвейером инструкция выполняется каждый тактовый цикл, несмотря на то, что для выполнения любой отдельной инструкции требуется много тактовых циклов.

Ветви — враги конвейеров. ЦП не может начать работу над следующей инструкцией, если он не знает, какая инструкция будет выполнена следующей. Он должен завершить вычисление условия ветвления, прежде чем он сможет это узнать. Чтобы справиться с этим, конвейерные ЦП также оснащены *предикторами ветвления* . Он делает предположение о том, какая ветвь будет выбрана, и начинает выполнять инструкции на этой ветви. Первоначально прогноз делается с использованием статической эвристики, а затем эти прогнозы улучшаются путем обучения на предыдущем поведении . Это даже включает прогнозирование количества итераций цикла, чтобы конечная итерация не была неверно предсказана.

Неправильно предсказанная ветвь имеет два ужасных последствия. Во-первых, весь прогресс на неправильной ветви должен быть отброшен. Во-вторых, конвейер будет очищен, и ЦП будет неэффективен, пока конвейер не заполнится инструкциями на правильной ветви. При достаточно глубоком конвейере может быть легко **более эффективно вычислить и отбросить ненужный результат, чем избегать его вычисления в**

первую очередь . Устранение ветвей означает устранение опасностей неправильного предсказания.

Еще одной опасностью для конвейеров являются *зависимости* . Если инструкция зависит от результата предыдущей инструкции, ей, возможно, придется ждать, пока предыдущая инструкция не достигнет достаточного прогресса, прежде чем она сможет завершить один из своих этапов. Это известно как *задержка конвейера* , и это важное соображение при проектировании архитектуры набора инструкций (ISA).

Например, в архитектуре x86-64 сохранение 32-битного результата в 64-битном регистре автоматически очистит верхние 32 бита этого регистра. Любое дальнейшее использование этого целевого регистра не может зависеть от предыдущих инструкций, поскольку все биты были установлены. Эта конкретная оптимизация была упущена в конструкции i386: запись 16-битного результата в 32-битный регистр оставляет верхние 16 бит нетронутыми, создавая ложные зависимости.

Опасности зависимости смягчаются с помощью *выполнения вне очереди* . Вместо того, чтобы выполнять две зависимые инструкции друг за другом, что привело бы к остановке, ЦП может вместо этого выполнить независимую инструкцию дальше между ними. Хороший компилятор также попытается распределить зависимые инструкции в своем собственном планировании инструкций.

Эффекты внеочередного выполнения обычно не видны для одного потока, где все будет выглядеть выполненным по порядку. Однако, когда несколько процессов или потоков могут получить доступ к одной и той же памяти, внеочередное выполнение может наблюдаться . Это одна из многих проблем написания многопоточного программного обеспечения .

Целью моего декодера UTF-8 было отсутствие ветвлений, но была одна интересная опасность зависимости, которую ни GCC, ни Clang не смогли разрешить самостоятельно. Подробнее об этом позже.

Что такое UTF-8?

Не вдаваясь в историю, можно в общем смысле рассматривать UTF-8 как метод кодирования последовательности 21-битных целых чисел (*кодовых точек*) в поток байтов.

- Более короткие целые числа кодируются в меньшее количество байтов, чем более длинные целые числа. Необходимо выбрать самую короткую доступную кодировку, то есть для данной последовательности кодовых точек существует одна каноническая кодировка.
- Некоторые кодовые точки находятся вне пределов: *суррогатные половины* . Это кодовые точки `U+D800`до `U+DFFF`. Суррогаты используются в UTF-16 для представления кодовых точек выше U+FFFF и не служат никакой цели в UTF-8. Это имеет интересные последствия для строк псевдо-Unicode, таких «широких» строк в

Win32 API, где суррогаты могут казаться непарными. Такие последовательности не могут быть законно представлены в UTF-8.

Принимая во внимание эти два правила, весь формат можно представить в виде следующей таблицы:

```
length byte[0]  byte[1]  byte[2]  byte[3]
1      0xxxxxxx
2      110xxxxx 10xxxxxx
3      1110xxxx 10xxxxxx 10xxxxxx
4      11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Заполнители `x`— это биты закодированной кодовой точки.

UTF-8 обладает некоторыми действительно полезными свойствами:

- Он обратно совместим с ASCII, в котором никогда не использовался старший бит.
- Порядок сортировки сохраняется. Сортировка набора последовательностей кодовых точек имеет тот же результат, что и сортировка их кодировки UTF-8.
- Никаких дополнительных нулевых байтов не вводится. В C мы можем продолжать использовать `char`буферы с нулевым завершением, часто даже не осознавая, что они содержат данные UTF-8.
- Он самосинхронизируется. Начальный байт никогда не будет ошибочно принят за байт продолжения. Это позволяет выполнять побайтовый поиск подстрок, что означает, что функции, не поддерживающие UTF-8, такие как `strstr(3)`continue, будут работать без изменений (за исключением проблем с нормализацией). Это также позволяет однозначно восстанавливать поврежденный поток.

Простой подход к декодированию может выглядеть примерно так:

```
unsigned char *
utf8_simple(unsigned char *s, long *c)
{
    unsigned char *next;
    if (s[0] < 0x80) {
        *c = s[0];
        next = s + 1;
    } else if ((s[0] & 0xe0) == 0xc0) {
        *c = ((long)(s[0] & 0x1f) <<  6) |
             ((long)(s[1] & 0x3f) <<  0);
        next = s + 2;
    } else if ((s[0] & 0xf0) == 0xe0) {
        *c = ((long)(s[0] & 0x0f) << 12) |
             ((long)(s[1] & 0x3f) <<  6) |
             ((long)(s[2] & 0x3f) <<  0);
        next = s + 3;
    } else if ((s[0] & 0xf8) == 0xf0 && (s[0] <= 0xf4)) {
        *c = ((long)(s[0] & 0x07) << 18) |
             ((long)(s[1] & 0x3f) << 12) |
             ((long)(s[2] & 0x3f) <<  6) |
             ((long)(s[3] & 0x3f) <<  0);
        next = s + 4;
    } else {
        *c = -1; // invalid
        next = s + 1; // skip this byte
    }
    if (*c >= 0xd800 && *c <= 0xdfff)
        *c = -1; // surrogate half
    return next;
}
```

Он ответвляется от старших битов ведущего байта, извлекает все эти xбиты из каждого байта, объединяет эти биты, проверяет, является ли это суррогатной половиной, и возвращает указатель на следующий символ. (Эта реализация не *проверяет* правильность старших двух битов каждого байта продолжения.)

Процессор должен правильно предсказать длину кодовой точки, иначе он подвергнется опасности. Неправильная догадка остановит конвейер и замедлит декодирование.

В реальном тексте это, вероятно, не является серьезной проблемой. Для английского языка закодированная длина почти всегда составляет один байт. Однако даже для неанглийских языков текст обычно сопровождается разметкой из диапазона символов ASCII , и, в целом, закодированные длины будут по-прежнему иметь согласованность. Как я уже сказал, ЦП предсказывает ветвления на основе предыдущего поведения программы, так что это означает, что он временно узнает

некоторые статистические свойства языка, который активно декодируется. Довольно круто, да?

Устранение ветвей из декодера обходит любые проблемы с неправильным предсказанием закодированных длин. Только ошибки в потоке могут вызвать остановку. Поскольку это, вероятно, необычный случай, предсказатель ветвей будет очень успешным, постоянно предсказывая успех. Это один оптимистичный ЦП.

Декодер без ветвлений

Вот интерфейс моего декодера без ветвлений:

```
void *utf8_decode(void *buf, uint32_t *c, int *e);
```

Я выбрал `void *`для буфера, чтобы не волновало, какой тип был фактически выбран для представления буфера. Это может быть `uint8_t`, `char`, `unsigned char`, и т. д. Не имеет значения. Кодер обращается к нему только как к байтам.

С другой стороны, с этим интерфейсом вы вынуждены использовать `uint32_t` для представления кодовых точек. Вы всегда можете изменить функцию в соответствии с вашими собственными потребностями.

Ошибки возвращаются в `e`. Это ноль в случае успеха и не ноль при обнаружении ошибки, без какого-либо особого смысла для различных значений. Условия ошибок смешиваются в этом целом числе, поэтому ноль просто означает отсутствие ошибки.

Вот тут вы могли бы обвинить меня в небольшом «мошенничестве». Вызывающий, вероятно, хочет проверить наличие ошибок, поэтому *ему* придется перейти на `e`. Похоже, я просто протащил ветки за пределы декодера.

Однако, как я уже указывал, если вы не ожидаете большого количества ошибок, реальная стоимость — это ветвление по закодированным длинам. Более того, вызывающий объект мог бы вместо этого накапливать ошибки: подсчитывать их или делать ошибку «липкой», объединяя все `e`значения с помощью ИЛИ. Ни один из этих вариантов не требует ветвления. Вызывающий объект мог бы декодировать огромный поток и проверять наличие ошибок только в самом конце. Единственным ветвлением был бы основной цикл («мы уже закончили?»), который легко предсказать с высокой точностью.

Первое, что делает функция, — извлекает закодированную длину следующей кодовой точки:

```
static const char lengths[] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 3, 3, 4, 0
};

unsigned char *s = buf;
int len = lengths[s[0] >> 3];
```

Возвращаясь к таблице UTF-8 выше, только самые старшие 5 бит определяют длину. Это 32 возможных значения. Нули предназначены для недопустимых префиксов. Это позже приведет к установке бита в `e`.

Имея на руках длину, можно вычислить положение следующей кодовой точки в буфере.

```
unsigned char *next = s + len + !len;
```

Первоначально это выражение было возвращаемым значением, вычисляемым в самом конце функции. Однако, после проверки ассемблерного вывода компилятора, я решил переместить его выше, и результатом стало солидное повышение производительности. Это потому, что оно распределяет зависимые инструкции. Поскольку адрес следующей кодовой точки известен так рано, инструкции, которые декодируют следующую кодовую точку, могут начать выполняться раньше .

Причина в `!len`том, что указатель продвигается на один байт даже при возникновении ошибки (длина равна нулю). Добавление этого `!len`на самом деле довольно затратно, хотя я не смог понять, почему.

```
static const int shiftc[] = {0, 18, 12, 6, 0};

*c  = (uint32_t)(s[0] & masks[len]) << 18;
*c |= (uint32_t)(s[1] & 0x3f) << 12;
*c |= (uint32_t)(s[2] & 0x3f) <<  6;
*c |= (uint32_t)(s[3] & 0x3f) <<  0;
*c >>= shiftc[len];
```

Это считывает четыре байта независимо от фактической длины. Избегание выполнения чего-либо является ветвлением, так что тут ничего не поделаешь. Ненужные биты сдвигаются в зависимости от длины. Это все, что нужно для декодирования UTF-8 без ветвления.

Одним из важных последствий постоянного чтения четырех байтов является то, что **вызывающий *должен* дополнить буфер нулями по крайней мере до четырех байтов** . На практике это означает заполнение всего буфера тремя байтами в случае, если последний символ представляет собой один байт.

Для обнаружения ошибок заполнение должно быть равно нулю. В противном случае заполнение может выглядеть как допустимые байты продолжения.

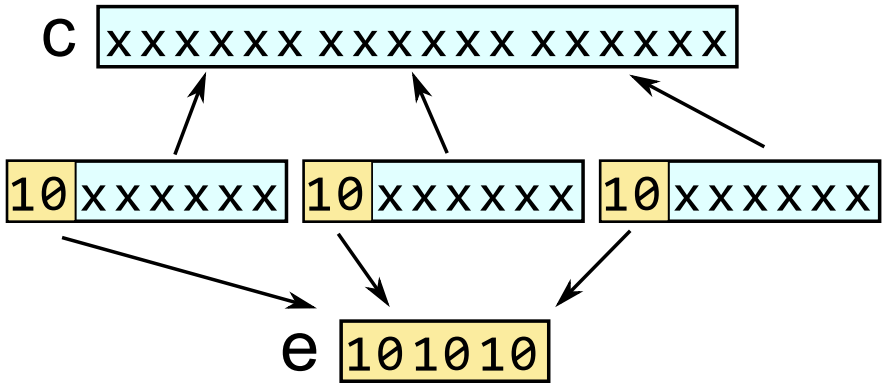
```
static const uint32_t mins[] = {4194304, 0, 128, 2048, 65536};
static const int shifte[] = {0, 6, 4, 2, 0};

*e  = (*c < mins[len]) << 6;
*e |= ((*c >> 11) == 0x1b) << 7;  // surrogate half?
*e |= (s[1] & 0xc0) >> 2;
*e |= (s[2] & 0xc0) >> 4;
*e |= (s[3]       ) >> 6;
*e ^= 0x2a;
*e >>= shifte[len];
```

Первая строка проверяет, использовалась ли самая короткая кодировка, устанавливая бит, `e`если это не так. Для длины 0 это всегда не удается.

Вторая строка проверяет суррогатную половину, проверяя определенный префикс.

Следующие три строки накапливают два старших бита каждого байта продолжения в `e`. Каждая должна быть битами `10`. Эти биты «сравниваются» с `101010`(`0x2a`) с помощью XOR. XOR очищает эти биты, если они точно совпадают.



Наконец, не имеющие значения биты префикса продолжения выдвигаются.

Цель

Моя главная — и совершенно произвольная — цель состояла в том, чтобы превзойти производительность декодера на основе DFA Бьорна Хёрмана . При благоприятных (и искусственных) условиях тестирования я имел умеренный успех. Вы можете попробовать это на своей системе, клонировав репозиторий и запустив `make bench`.

С GCC 6.3.0 на i7-6700 мой декодер примерно на 20% быстрее, чем декодер DFA в бенчмарке. С Clang 3.8.1 он всего на 1% быстрее.

Обновление : Бьёрн указал , что на его сайте есть более быстрый вариант его декодера DFA. Он всего на 10% медленнее, чем декодер без ветвлений с GCC, и на 20% быстрее, чем декодер без ветвлений с Clang. Так что, в некотором смысле, он все еще быстрее в среднем, даже на бенчмарке, который предпочитает декодер без ветвлений.

Тест работает очень похоже на мою перестрелку PRNG (например `alarm(2)`). Сначала буфер заполняется случайными данными UTF-8, затем декодер декодирует их снова и снова, пока не сработает сигнализация. Измерение — это количество декодированных байтов.

Количество ошибок выводится в конце (всегда 0), чтобы заставить ошибки действительно проверяться для каждой кодовой точки. В противном случае хитрый компилятор пропускает проверку ошибок из декодера без ветвлений, заставляя его казаться намного быстрее, чем он есть на самом деле — серьезное разочарование, когда я заметил свою ошибку. Поскольку другой декодер — это DFA, и проверка ошибок встроена в его граф, компилятор не может на самом деле пропустить свою проверку ошибок.

Я назвал это «благоприятным», потому что декодируемый буфер не является чем-то естественным. Каждый раз, когда генерируется кодовая точка, сначала выбирается длина равномерно: 1, 2, 3 или 4. Затем генерируется кодовая точка, которая кодирует эту длину. **Равномерное распределение длин значительно благоприятствует декодеру без ветвлений** . Случайное распределение подавляет предсказание ветвлений. Реальный текст имеет гораздо более благоприятное распределение.

```
uint32_t
randchar(uint64_t *s)
{
    uint32_t r = rand32(s);
    int len = 1 + (r & 0x3);
    r >>= 2;
    switch (len) {
        case 1:
            return r % 128;
        case 2:
            return 128 + r % (2048 - 128);
        case 3:
            return 2048 + r % (65536 - 2048);
        case 4:
            return 65536 + r % (131072 - 65536);
    }
    abort();
}
```

Учитывая странное требование нулевого заполнения входных данных и искусственные параметры бенчмарка, несмотря на предполагаемый 20%-

ный прирост скорости при GCC, мой декодер без ветвлений на практике ничем не лучше декодера DFA. Это просто другой подход. На практике я бы предпочел декодер DFA Бьёрна.

Обновление : Брайан Донлан выпустил декодер SIMD UTF-8 .

Обновление 2024 : NRK выпустила параллельный декодер извлечения .

Обновление 2025 : Чарльз Экман продолжил делиться кодером без ветвлений , что вдохновило меня попробовать его .