Енкодирање по основи 41

Мај 2015

1 Кодирање по основи 41 (Б41)

Б41 кодирање је инспирисано Ascii85 кодирањем. Замислите ово кодирање као “ASCII85 за реч од два уместо четири октета”.

1.1 Подсетник: Ascii85

Ascii85 је код који користи чињеницу да је $85 ** 5 > 2 ** 32$, док је $84 ** 5 < 2 ** 32$. Тако да, са 5 знакова који имају један од 85 могућих вредности, можемо да кодирамо 4 октета (код модерних рачунара отприлике сви бајтови су октети). Много података садржи више пута по 4 октета, али, чак и да не садржи, можете их допунити/додати. Стога, ово кодирање је прилично ефикасно - има вишак од 25% (мало више, ако морате да допуњујете/додајете).

Можемо звати четворо-бајтни елемент података и његов пето-знаковни приказ “(Ascii85) речи”.

Проблем са Ascii85 је да његов алфабет произилази из једноставне ASCII value − 33 формуле. То значи да користи наводнике и обрнуте косе црте, који се, у већини текстова, укључујући и већину програмских језика, користе за приказ ниски. Такође, Ascii85 има специјално правило за знак z - он представља “нула реч” (реч која се састоји од 4 бајта једнаких нули). Не морате га тако кодирати, али га морате декодирати (ако налетите на z). Дакле, ово је један специјалан случај који није баш фин.

Постоје друга кодирања са основом 85, као Z85 (ZeroMQ Base85) и RFC 1924, која реше проблеме алфабета дефинишући га као табелу која прескаче наводнике и обрнуте косе црте. Ово је занимљиво, али захтева две табеле (једну за кодирање и другу за декодирање). Код уграђених система, ово може да представља проблем, и у величини потребне (радне) меморије, и у времену обраде.

Такође, код уграђених система, некада је незгодно допуњавати податке како би садржали више пута по 4 октета, посебно за краће податке.

1.2 Основа 41 (Б41)

Значи, ми смо као Ascii85, али за два октета у једној речи. Опажамо да је $41 ** 3 > 2 **16$ и $40 ** 3 < 2 ** 16$. Гледајући на ASCII табелу, бирамо алфабет као ASCII value − 41. Овај алфабет не садржи наводнике или обрнуте косе црте, тако да је безбедан за ниске.

Ако подаци имају непаран број елемената… па, допуните их. Много је мање незгодно допунити једним бајтом него са “једним до три” бајта (и допуњавате три пута мање него са Ascii85). Ми не одређујемо како допуњујете, то препуштамо корисницима, али будите свесни да друга страна мора да зна да су подаци допуњени и како.

Требало би да је очигледно да је вишак Б41 кодирања 50% (за податке са непарним бројем октета, је нешто већи, процентуално зависно од величине конкретног податка, због додавања једног октета допуне).

1.3 Код

… је поприлично једноставан, што није чудно, с обзиром да смо дизајнирали кодирање са тим у виду:

```
void base41_encode(uint8_t const *input, size_t n, char *output)
{
    uint8_t const *p = input;
    char *s = output;

    assert(n % 2 == 0);

    for (p = input; p < input + n; p += 2) {
        int x = *p + 256 * p[1];
        *s++ = (x % 41) + 41;
        x /= 41;
        *s++ = (x % 41) + 41;
        *s++ = x/41 + 41;
    }
    *s = '\0';
}

void base41_decode(char const *input, uint8_t *output)
{
    char const *s = input;
    size_t n = strlen(s);
    uint8_t *p = output;
    int i;

    assert(n % 3 == 0);

    for (i = 0; i < n; i += 3) {
        int x = (*s - 41) + 41 * (s[1] - 41) + 41*41*(s[2] - 41)
            ;
        *p++ = x % 256;
        *p++ = x / 256;
        s += 3;
    }
}
```

Овај код претпоставља да корисник зна како да заузме потребну меморију за излаз (output). За изворни код који је мало више одбрамбен/параноичан, можда ћете хтети да проследите величину излаза и проверите или утврдите (assert)на почетку.

У декодирању, x / 256 је можда веће од 255 код лоше кодиране ниске, али, нас то не занима, с обзиром да га додељујемо октету, стога ће "вишак" бити избачен. Параноичан код би могао ово да провери и пријави грешку.

1.4 Лоше кодирање - слово није у алфабету

Декодирање ниске која има знакове који нису у Б41 алфабету није дефинисано. Извршење које је више параноично може да пријави грешку, док би извршење као ово изнад, могло само да га "остави на миру" и декодира нешто.

Али, ово није "недефинисано понашање", то је "недефинисано декодирање". Понашање је увек "декодирати сваку 'реч' од три слова на два октета". Нас просто не занима на које октете одлучите да декодирате, ако је Б41 ниска неисправна. Али, декодер не може да форматира диск, пише у меморију која није његова…

1.5 Лоше кодирање - лоше речи од три слова

Три слова Б41 која формирају кодирање за дво-октетни податак, се могу назвати "Base41 реч" и могу бити таква да њихова вредност премаши 65535.

Такве трословне "речи" нису исправне.

Декодер има слободу да их пријави као грешке или да декодира шта жели из тога. То јест, ограничавање на 65535 је океј, као и ограничавање до 0… Како год.

Дакле, ово је такође "недефинисано декодирање", а не "недефинисано понашање".

1.6 Лоше кодирање - ниска са дужином која није дељива са 3

Ово је потпуно дефинисано. Декодер ће само игнорисати вишак знакова.

То значи да, ако се пошаље четири до пет знакова, декодер ће декодирати само једну Б41 реч (прву). Ако се пошаље шест знакова, декодер ће декодирати две Б41 речи.

Тако да, ако је n дужина Б41 ниске, декодер ће декодирати n/3 речи (или (n/3)∗3 знака) где је / целобројно дељење.

1.7 Само ми покажи алфабет

Ако инсистирате:

```
 ) * + , – . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q
```

То је лепо поређано, али, можда је лакше овако разумети:

```
  ) * + , – . / 0 1 2
3 4 5 6 7 8 9 : ; <
= > ? @ A B C D E F
G H I J K L M N O P
Q
```

За оне који воле табеле:

value	symbol	value	symbol	value	symbol	value	symbol
0	)	10	3	20	=	30	G
1	*	11	4	21	>	31	H
2	+	12	5	22	?	32	I

3	,	13	6	23	@	33	J
4	–	14	7	24	A	34	K
5	.	15	8	25	B	35	L
6	/	16	9	26	C	36	M
7	0	17	:	27	D	37	N
8	1	18	;	28	E	38	O
9	2	19	<	29	F	39	P
40	Q						

1.8 Дискусија

1.8.1 Чему ово служи?

Добро је за кодирање бинарних података у JSON (или сличним) нискама.

Поприлично је брзо и користи веома мало програмске меморије и само мало стек меморије и нимало непроменљиве меморије. То га чини погодним за уграђене системе.

Лако је направити га у било ком језику.

Лако је додати га у ваш Javascript код. Javascript "модули" су хаос. Javascript их баш и не подржава и сва извођења истог су само конвенције. Наравно, данас су npm и различити алати који "граде" ваш Javascript у широкој употреби, али, и даље је неред. Неред који с времена на време можете почистити неким усисивачима. Много је лакше само ставити код који вам је потребан на саму страницу, али то постане проблем кад има много поменутог кода. Али, Б41 је само пар линија Javascript-а. Кад смо већ ту:

```
function base41_decode(input) {
    var rslt = []
    var n = input.length
    for (var i = 0; i < n; i += 3) {
        var x = (input.charCodeAt(i) - 41) + 41 * (input.
            charCodeAt(i+1) - 41) + 1681*(input.charCodeAt(i+2)
            - 41);
        rslt.push(x % 256, Math.floor(x / 256))
    }
    return rslt
}
function base41_encode(input) {
    var rslt = ""
    var n = input.length
    for (var i = 0; i < n; i += 2) {
        var x = input[i] + 256 * input[i+1];
        rslt = rslt.concat(String.fromCharCode((x % 41) + 41));
        x /= 41;
        rslt = rslt.concat(String.fromCharCode((x % 41) + 41),
            String.fromCharCode((x / 41) + 41))
    }
    return rslt
}
```

Опет, ово је краћа, "не параноична" верзија. Додајте провере по вољи.

1.8.2 Зашто не Б64, такође је безбедно за ниске?

Б64 је мало чудан:

- Постоје два Б64 кодирања, једно за електронску пошту, друго за URL-ове
- Постоје разне варијанте Б64 кодирања, са разликом у алфабету, допуњујућим знаковима, обради знака за крај реда…
- Многа изводјења претпоставе да је текст (а не бинарни садржај) оно што је кодирано/декодирано и пријаве грешку ако није
- Требају му табеле за кодирање и декодирање
- Ретко који подаци по дужини садрже више пута по 3 октета, осим случајно, тако да ће већину времена имати један или два допуњујућа знака на крају. За кратке податке ово ће можда бити важно.
- Код је чудан, комплексан и незгодан, са тим парчићима од 3 бајта…
- С обзиром да је незгодан, код такође није ни мали нити брз

1.8.3 Ово кодирање захтева дељење са 41, зар то није споро?

На неким процесорима са референтним извођењем, да. Наравно, на таквим процесорима би можда било боље користити неко друго кодирање, или пробати неку оптимизовану верзију.

Ако вам је брзина битна, најједноставније кодирање које можете користити је Б16 варијација, која је као Hex (Б16), само што уместо 0123456789ABCDEF користи ABCDEFGHIJKLMNOP алфабет (ASCII value - 65). Нема проблеме допуњавања, тако да је најједноставније што може да буде. Једини проблем је што има вишак од 100%.

Али, такође, у тексту испод обратите пажњу на трикове за ефикасно дељење са 41.

1.8.4 Зашто почети са АСЦИИ 41? А не, на пример, 48?

Очигледно, ово није случајно. Желимо да избегнемо наводнике, али такође и #, $ и % који се често користе приликом уметања у ниске у различитим програмским језицима.

У ствари, први ASCII знак који је безбедан за ниску би био вредност 40, знак (. Али, дељење са 41 одговара "основи", тако да ћемо то изабрати.

Истина је да је ASCII 41), тако да би можда било лепо то избећи, пошто смо већ избегли (. Са друге стране, како користимо само затворену заграду, избегавамо проблем у неким тумачењима ниски.

Постоји једна занимљива идеја да се за основу користи 48. Добра ствар код тога је да дељење са 48 може да буде постигнуто и дељењем са 16 (померање за 4 бита) затим дељењем са 3, што је посебан облик дељења које може бити ефикасније него дељење са 41, који је велики прост број. Тако да, код споријих процесора, посебно оних који немају делитељ у процесору, ово би вероватно било брже него дељење са 41.

Али:

- таквим процесорима би више одговарало Б32 или Б16 кодирање
- заправо, цака за ефикасно дељење са 3 је искористити множење, а то такође може да се користи и за дељење са 41. Тако да, осим ако би неко могао да искористи неки трик без множења за дељење са 3, немамо користи. Ово је даље анализирано испод
- 48 ** 3 је много веће него 2 ** 16, тако да се јавља осећај вишка
- не можемо да започнемо алфабет на ASCII 48, пошто би то "припојило" обрнуту косу црту. Можемо да започнемо на ASCII 42, што би се завршило на ASCII 90 што је Z, али нам се допада идеја "почетка од основе".

Ефикасно дељење са 3 Трик је помножити неким бројем који је производ 3 и неког степена двојке. Наравно, најбољи такав број је приказ 1/3 у бинарном запису фиксне дужине.

Дакле, за 32-битне процесоре (који су данас чести у свету уграђених система), ово је дељење са 3:

```
(x * 0x5556) >> 16;
```

а остатак је:

```
x - ((x * 0x5556) >> 16) * 3);
```

Стога, петља кодирања Б48 би била:

```
for (p = input; p < input + n; p += 2) {
    int x = *p + 256 * p[1];
    int q = ((x>>4) * 0x5556) >> 16;
    *s++ (x - q*48) + 42;
    x = q;
    q = ((x>>4) * 0x5556) >> 16;
    *s++ (x - q*48) + 42;
    x = q;
    q = ((x>>4) * 0x5556) >> 16;
    *s++ q + 42;
}
```

За 16-битне процесоре, морате да изводите више трикова, и тренутно немамо решење овде. Али можете пробати овај код на 16-битном процесору, његова имитација 32-битне аритметике можда буде бржа него његово 16-битно дељење.

Наравно, ово није баш паметно, како дељење са 48 може бити написано као (x*0x5556) >> 20, тако да би паметнија петља била:

```
for (p = input; p < input + n; p += 2) {
    int x = *p + 256 * p[1];
    int q = (x * 0x5556) >> 20;
    *s++ (x - q*48) + 42;
    x = q;
    q = (x * 0x5556) >> 20;
    *s++ (x - q*48) + 42;
    *s++ ((x * 0x5556) >> 20) + 42;
}
```

Ефикасно дељење са 41 Заправо, исти трик може да се користи за 41. То јест, приближно дељењу са 41 је:

```
(x * 0x63e7) >> 20;
```

и остатак је:

```
x - ((x * 0x63e7) >> 20) * 41);
```

и петља кодирања је:

```
for (p = input; p < input + n; p += 2) {
    int x = *p + 256 * p[1];
```

```
    int q = (x * 0x63e7) >> 20;
    *s++ = (x - q*41) + 41;
    x = q;
    q = (x * 0x63e7) >> 20;
    *s++ = (x - q*41) + 41;
    *s++ = ((x * 0x63e7) >> 20) + 41;
}
```

Значи, у ствари је ефикасније него “наивно” дељење са 48 и ефикасно колико и “паметно” дељење са 48.

Успут, 1/41 је 0x063e70…, тако да, с обзиром да су последњих четири показана бита 0, има фино својство: можемо искористити првих пет хексадекадних цифара (20 бита), и тиме у ствари добијемо прецизност 6 хексадекадних цифара (24 бита)! То јест, с обзиром да су та последња 4 бита сви 0, не утичу на прецизност, ако сте у реду са 24-битном прецизношћу. Овде, пошто је грешка врло мала, чак и после много рачунања, а код нас их неће бити пуно, неће да нарасте толико да нам направи проблем.

Ова грешка у дељењу је много боља него она за 1/40 и мало боља него она за 1/42. Такође, то је пука случајност, основа 41 није била изабрана због тога. Међутим, током истраживања о “оптимизацијама дељења”, закључили смо да је то добра ствар, што учвршћује наш избор основе 41.

Стога, ако сте на 32-битном (или 64-битном) процесору, користите ову петљу (осим ако имате одличан процесор који има брз делитељ у процесору). За 64-битне процесоре, неке додатне оптимизације су могуће, али то је ван оквира наше анализе (која се ограничава на “зашто основа 41 а не 48”).

1.9 Алфабет основе 41 има < i > тако да је то лоше за XML

Ми не волимо XML, па нас то не занима.

Шалу на страну, кодирање бинарних података у XML-у свакако није најбоља идеја, Ако баш желите, користите Б64, или неки ASCII85 дијалекат одговарајућ XML-у.

Такође, ако користите XML, веома је вероватно да нисте на ограниченом уграђеном систему, тако да вам додатне компликације кодирања “вишег” од Б41 неће бити проблем.

На крају крајева, додајте кодиране податке у XML атрибут (ниску), на пример:

```
<some-tag base41="ABC" />
```

1.10 Занимљиви примери?

Ево неколико занимљивих примера кодирања и декодирања.

Декодирање:

Бинарно	Ниска	напомена
[49,49] (ASCII “11”)	”/=0”	Б41 је математички тачна, пошто 11 није једнако 0
[78,79] (ASCII “NO”)	”0,5”	“не” значи “можда”

Кодирање:

Ниска	Бинарно	напомена
”861”	[172,54] Latin1 NOT“6”	још математичког доказа да је Б41 тачно, на енглеском може да се чита као 861 није 6
”:—P”	недозвољено (било би [204,256])	Base41 је уљудно, не дозвоља Вам да се плазите људима