Arquivos Binários

Canoi Gomes

23 de Junho de 2024

Conteúdo

1 Introdução

Você sabe como funciona um arquivo binário? Aqueles arquivos que ao tentarmos abrir num editor de textos qualquer, aparece um monte de símbolos que mal dá para entender do que se trata. Bem, e se eu te disser que tem uma lógica por trás deles? E isso é mais ou menos o que quero abordar nessas linhas.

1.1 O que é um arquivo binário?

Simplificando bastante, um arquivo binário nada mais é que um arquivo de texto, porém formatado de uma maneira bem específica onde o programa que for fazer uso daquele arquivo deve saber como é a sua estrutura interna. Quando digo que não se difere muito do arquivos de texto, é porque temos isso com eles também, se criamos um arquivo JSON ou XML, também temos uma estrutura que se for bagunçada é o suficiente para corromper o arquivo. Da mesma forma ambos são usados para guardar algum tipo de informação. A principal diferença é que arquivos de texto estão preocupados com os caracteres do arquivo, já um arquivo binário se preocupa com como os bytes estão posicionados, e até mesmo cada bit.

Alguns formatos de arquivos binários que posso citar são o .zip, .bmp, .wav, .png, que como vocês podem ver são usados para guardar diferentes tipos de dados. Até mesmo os nossos arquivos executáveis (PE no Windows (.exe) e ELF no Linux) são binários, e o nosso sistema operacional que sabe como interpretá-los. Mais pra frente quando eu for mostrar a estrutura de alguns formatos mais populares (.bmp, .tar, .wav, ...), você vai ver que é possível até mesmo interpretar o que está dentro de arquivos binários executáveis.

2 Criando um Formato de Arquivo Binário

Beleza, mas por onde podemos começar se queremos escrever um formato de arquivo binário e um programa que seja capaz de o interpretar?

Bom, veja esses exemplos:

file.json:

```
{
    "magic": "MF",
    "name": "Ze Brasileiro",
    "idade": 25,
    "vivo": true
}
```

file.bin:

```
MFZe Brasileiro@$
```

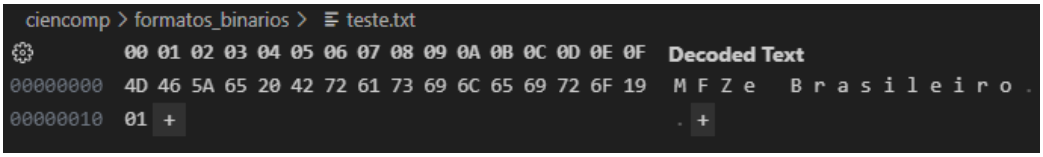
Coloquei esses símbolos no final só para representar que algo foi escrito naqueles campos, mas imagine que ali o @ representa 25 e $ representa 1. Mas perceba que ambos guardam o mesmo tipo de informação, dados sobre nome, idade, se a pessoa está viva, e o magic ali é pra representar um **magic number**, é comum em formatos de arquivos binários você ter um desses, ele serve basicamente para validação do tipo. Por exemplo, caso você crie um programa que vá ler especificamente esse seu formato, então quando um usuário for tentar abrir um arquivo qualquer, você checa se a magic desse arquivo é de fato "MF", e caso contrário você retorna uma mensagem de erro dizendo que o tipo é inválido. É um jeito mais seguro que só checar pela extensão do arquivo, por exemplo.

2.1 Tabela ASCII

Para entender o que aconteceu com os valores de Idade (25) e Vivo (true, ou 1), precisamos entender como funciona a construção de uma string e o que é uma

tabela ASCII.

Vamos analisar nosso arquivo usando um editor hexadecimal:



Perceba como cada caractere tem um valor hexadecimal correspondente. Para entender esses valores você pode conferir a tabela ASCII abaixo.

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

asciicharstable.com

Então olhando com mais calma:

- 0x4D e 0x46: Representa o nosso magic number, **4D = M** e **46 = F**
- 0x5A ao 0x6F: Representa o nosso nome, se pegarmos novamente na tabela ASCII vamos ver que **5A = Z** e **6F = o**
- 0x19 e 0x01: Essa que é a parte interessante, se pegarmos na tabela vamos ver que esses códigos representam caracteres especiais. Caracteres esses

que muito provavelmente não serão exibidos no editor de texto. Porém o que nos interessa não é o caractere que o código representa, e sim seu valor. 0x19 é o valor hexadecimal para 25, ou seja, a idade que nós colocamos para a pessoa. Igualmente o 0x01, o que nos interessa é somente o seu valor (true nesse caso).

2.2 Cabeçalho

Tá bom, mas como que eu represento isso no meu código? É bem comum utilizar algo chamado cabeçalho, que vai ter uma representação exata dos bytes utilizados por cada campo. Eu vou começar alterando um pouco o formato do meu tipo. No cabeçalho não vou guardar as informações relativas ao nome, e sim qual o tamanho do nome. E o dado do nome de fato, eu escrevo depois do meu cabeçalho. Seria algo mais ou menos assim:

```
MF#@$Ze Brasileiro
```

Ignore os simbolos de **#, @ e $** por agora, só saiba que eles representam respectivamente o [tamanho do nome][idade][vivo ou morto].

Representando em uma struct em C eu vou ter isso aqui:

```
struct Header {
    char magic[2];
    char name_size;
    char age;
    char alive;
};
```

Listing 1: Cabeçalho em C

Mas por que eu fiz isso? Bem, como estava anteriormente, com o nome vindo logo depois do magic number, um trabalho a mais é necessário para se interpretar esse arquivo. Seria necessário ler inicialmente os dois bytes, e depois começar

a ler caractere por caractere do nome até chegar em um número. Com isso eu não consigo identificar facilmente o tamanho do nome de cara, preciso fazer essa leitura byte a byte e pode dificultar um pouco no uso no programa.

Uma convenção bem comum é setar um tamanho fixo de bytes para guardar uma string.

```
struct Header {
    char magic[2];
    char name[32];
    char age;
    char alive;
};
```

Listing 2: Cabeçalho com tamanho fixo de string

Ou seja, seu nome não pode ser maior que 32 bytes, porém pode ser menor. Só que nesse caso, mesmo que o nome seja menor que 32 bytes, a informação da idade não vem logo depois do nome, ela começa somente no byte 34 (2 do magic + 32 do nome), pois o resto dos bytes será preenchido (provavelmente com 0). Teria algo assim:

```
MFZe Brasileiro0000000000000000000@$
```

Novamente, isso é só uma representação, é muito provável que no seu editor de textos você não veja nenhum símbolo após o nome.

2.3 Escrevendo em um arquivo binário

Mas vamos para o código em C.

```
#include <stdio.>
#include <string.h>

struct Header {
```

```
5       char magic[2]; // magic number "MF"
6       char name_size; // quantidade de bytes que tem meu nome
7       char age; // idade
8       char alive; // vivo (1 para true, 0 para false)
9   };
10
11  int main(int argc, char** argv) {
12      FILE* fp = fopen("file.bin", "wb"); // abrindo o arquivo
        file.bin em modo de escrita binária
13      const char* name = "Ze Brasileiro";
14      struct Header h;
15      h.magic[0] = 'M';
16      h.magic[1] = 'F';
17      h.name_size = strlen(name); // pego tamanho do nome em bytes
18      h.age = 25;
19      h.alive = 1;
20      fwrite(&h, 1, sizeof(h), fp); // escrevo o cabeçalho no
        inicio do arquivo
21      fwrite(name, 1, h.name_size, fp); // escrevo o nome logo em
        seguida
22      fclose(fp);
23      return 0;
24  }
```

Listing 3: Exemplo de escrita em C

Aqui não tem muito mistério. Eu preencho meu cabeçalho com os dados que quero utilizar e escrevo em um arquivo (lembre-se de abrir o arquivo no modo de escrita binária, ou **wb**), e logo em seguida escrevo o nome.

2.4 Lendo de um arquivo binário

Bom, e caso eu queira ler esse arquivo que eu acabei de criar. Continua sendo bem simples, a gente só precisar usar o mesmo cabeçalho, e dessa vez ao invés de abrir o arquivo no modo de escrita binária (**wb**), irei abrir no modo de leitura binária (**rb**):

```c
#include <stdio.>
#include <string.h>

struct Header {
    char magic[2]; // magic number "MF"
    char name_size; // quantidade de bytes que tem meu nome
    char age; // idade
    char alive; // vivo (1 para true, 0 para false)
};

int main(int argc, char** argv) {
    FILE* fp = fopen("file.bin", "rb"); // abrindo o arquivo file.bin em modo de leitura binária
    struct Header h;
    fread(&h, 1, sizeof(h), fp); // leio o cabeçalho no inicio do meu arquivo
    if (h.magic[0] != 'M' || h.magic[1] != 'F') return -1; // formato inválido
    char name[h.name_size+1];
    fread(name, 1, h.name_size, fp); // em seguida leio [name_size] bytes, que é o tamanho do meu nome
    name[h.name_size] = '\0';

    printf("nome: %s\n", name);
    fclose(fp);
    return 0;
```

```
}
```

Listing 4: Exemplo de leitura em C

2.5 Formato Binário Legível

Só para finalizar o assunto, também é possível construirmos um formato binário que seja legível/modificável por um editor de textos comum. Você só tem que se atentar com a maneira que seu programa interpreta o arquivo. Se eu quiser fazer algo assim, por exemplo:

```
MF13251Ze Brasileiro
```

É preciso notar que cada um dos números representam um caractere a parte. Ou seja, o 13 nós precisamos tratar como 1 e 3 tendo bytes separados, a mesma coisa pro 25. Poderia fazer um cabeçalho assim, por exemplo:

```
struct Header {
    char magic[2];
    char name_size[2];
    char age[2];
    char alive;
};
```

Mas para usar os valores reais é preciso converter de string para número:

```
13
1 (0x49 em ASCII) e 3 (0x51 em ASCII)
value = (1 - 0x48) * 10 + (3 - 0x48);
```

3 Considerações

Acho que é isso que tinha para escrever por agora, é só uma pincelada no assunto para entender como funciona a estrutura de um arquivo binário. Como falei

anteriormente, nos próximos textos vai ficar mais fácil de entender isso quando estivermos lidando com formatos de arquivo binário mais consolidados.