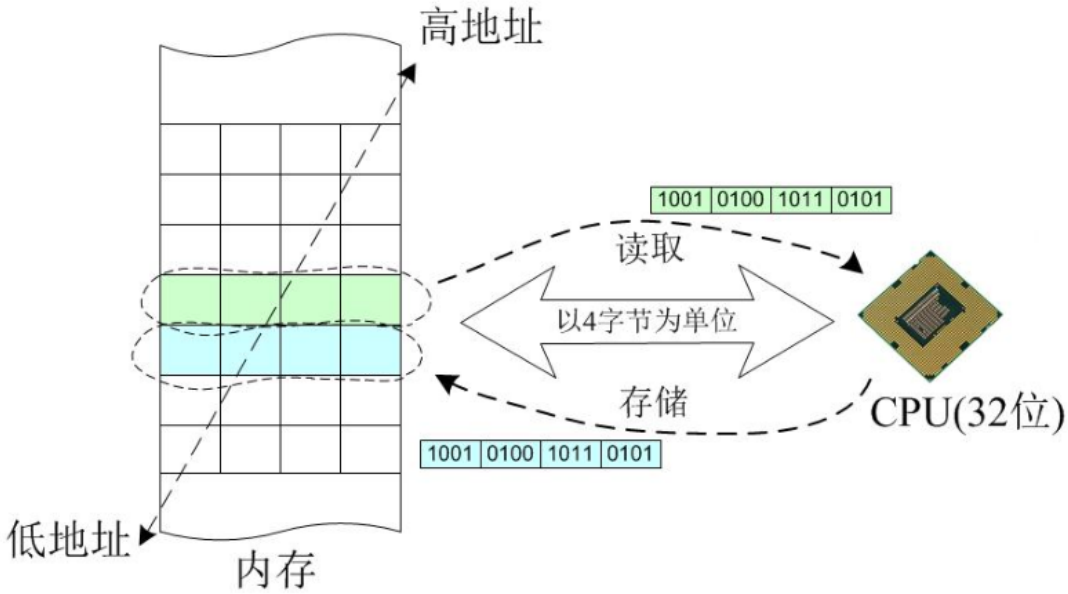
CPU 字长

字长就是指CPU（处理器）在执行一条指令时，最大一个运算能力，这个运算能力由两部份决定一个就是处理器本身的运算能力， 另一个就系统的字长，比如常见的32位/64位系统，如果使用32位系统那么在处理数据的时候每一次最对多可以处理32位的数据（4字节） 。



地址对齐

每一款不同的处理器，存取内存数据都会有不同的策略，如果是 32 位的 CPU，一般来讲他在存取内存数据的时候，每次至少存取 4 个字节（即 32 位），也就是按 4 字节对齐来存取的。换个角度讲：CPU 有这个能力，他能一次存取 4 个字节。

接下来我们可以想到，为了更高效地读取数据，编译器会尽可能地将变量塞进一个 4 字节单元里面，因为这样最省时间。如果变量比较大，4 个字节放不下，则编译器会尽可能地将变量塞进两个 4 字节单元里面，反正一句话：两个坑能装得下的就绝不用三个坑。这就是为什么变量的地址要对齐的最根本原因。

以一个 double 型变量为例，double 型变量占 8 个字节，以下是地址未对齐时的情况：

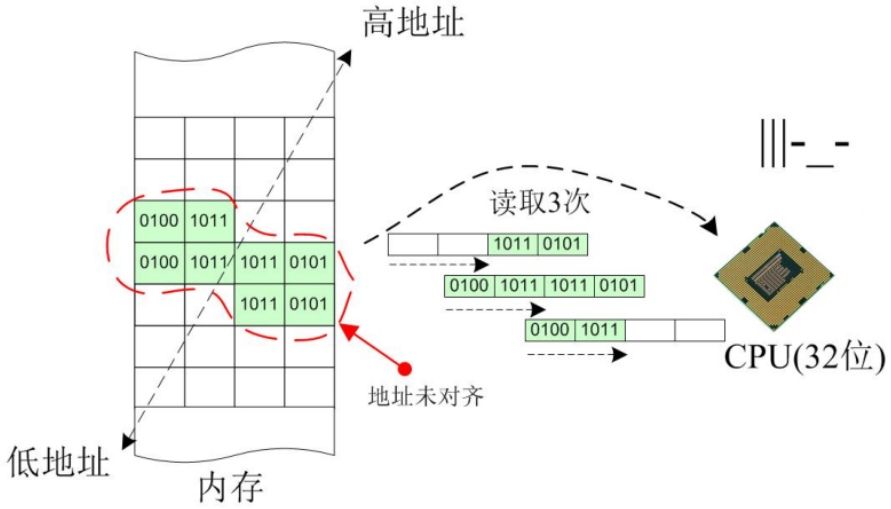


图 2-62 地址没对齐的数据

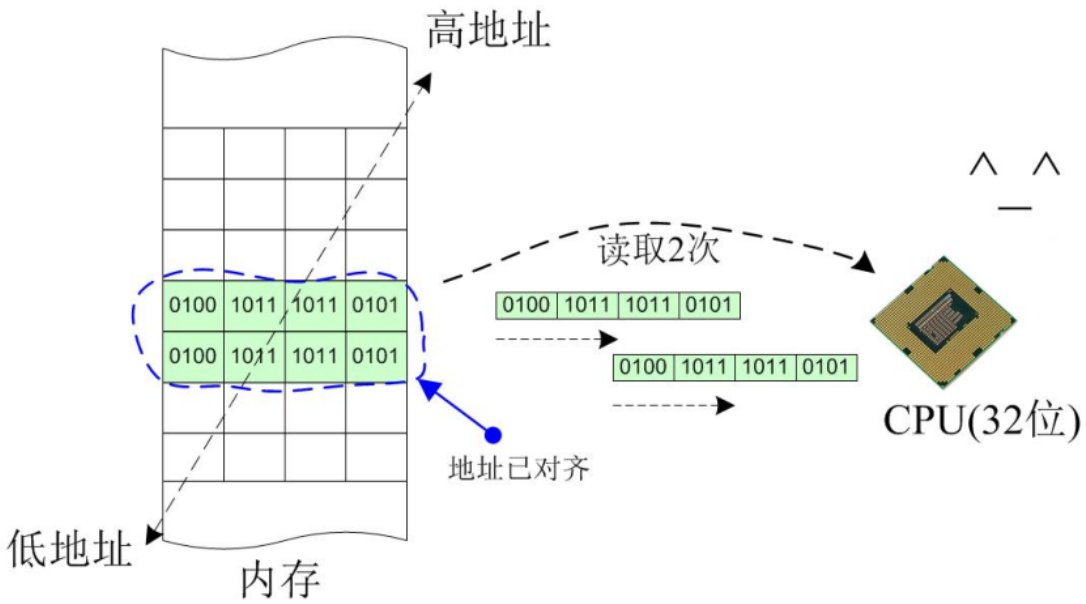


图 2-63 地址对齐的数据

所谓地址对齐主要思想：

尽可能提高CPU的运行效率（仅能能的少取读取/写入内存）

一个数据如能使用一个单元来存放感觉对不使用两个， 两个能放绝对不用三个

普通变量的M值

概念： 一个数据它的大小是固定的（比如整型），如果这个数据他所存放的地址能够被某一个数所整除（4）那么这个数就成为当前数据的M值。

可以根据具体的系统字长以及数据的大小可以计算得出M值。

例子：

```
int i ;  // i 占用 4 字节， 如果i 存放在能被4整除的地址下则是地址对齐， 因此他的M值为4
char c ;// c占用 1 字节， 如果c 存放在能被1整除的地址下则是地址对齐， 因此他的M值为1
short s ;// s 占用 2 字节， 如果s 存放在能被2整除的地址下则是地址对齐， 因此他的M值为2
double d ; // d 占用 8 字节， 如果d 存放在能被4整除的地址下则是地址对齐， 因此他的M值为4
float f ;// f 占用 4 字节， 如果f 存放在能被4整除的地址下则是地址对齐， 因此他的M值为4

i:0x7fffc900a298
s:0x7fffc900a296
c:0x7fffc900a295
d:0x7fffc900a2a0
f:0x7fffc900a29c
```

注意：

如果一个变量的大小超过4 （8/16/32） M值则按4计算即可

手残干预M值：

```
char c  __attribute__((aligned(16))) ;
```

注意：

__attribute__ 机制是GNU特定语法，属于C语言标准的拓展。

__attribute__ 前后都有个两个下划线_

__attribute__ 右边由两对小括号 (( ))

__attribute__ 还支持其它的设置.....

一个变量他的M值只能提升不允许降低， 只能是2的N次幂

结构体的M值

- 结构体中有多个成员，取决于成员中M值最大的成员。
- 结构体的地址，必须能被结构体的M值整除
- 结构体的尺寸，等于成员中宽度最宽成员的倍数

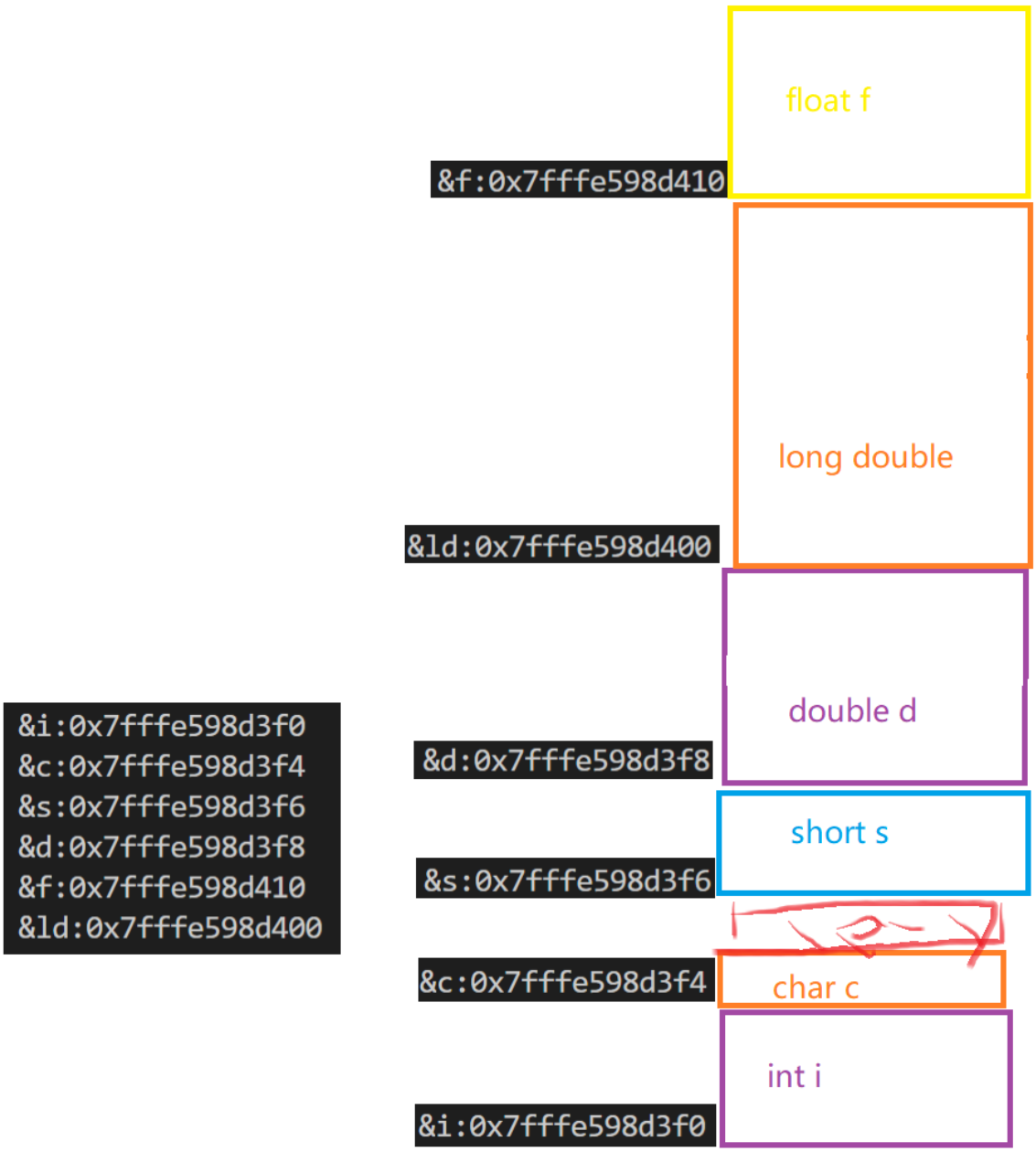
例子：

```
typedef struct node
{
    int i ;     // 4
    char c  ;   // 1
    short s ;   // 2
    double d ;  // 8
    long double ld ;  // 16    由于16是最宽的，因此结构体的大小不许能被16整除
    float f ;  // 4
}Node;
```

以上结构体在64位系统中：

大小 为 48 --> 能被 long double整除

地址值能被M值4整除



可移植性

对于可移植的结构体来说一定要解决好该结构体在不同的操作系统（位数）如何统一 该结构体的大小。

方法有两个：

方法1 ：

直接使用attribute 进行压实， 每一个成员之间没有留任何的空隙。

```
typedef struct node
{
    int i ;
    char c  ;
```

```
    short s ;
    double d ;
    long double ld ;
    char kk ;
    float f ;
}__attribute__(( packed ));
```

方法2：

对每一个成员进行压实

```
struct node
{
    int i __attribute__((aligned(4))) ;
    char c  __attribute__((aligned(1)));
    short s __attribute__((aligned(2)));
    double d __attribute__((aligned(4)));
    long double ld __attribute__((aligned(4))) ;
    char kk __attribute__((aligned(1))) ;
    float f  __attribute__((aligned(4)));
};
```

注意：

结构体的大小取决于多个因素

- 地址对齐，M值的问题， 默认情况下结构体的大小为成员中最大的倍数
- 结构体内部的每一个成员的大小都与系统的位数有关
- 如果需要实现可移植性，结构体的每一个成员需要使用可移植类型+attribute机制对结构体进行压实