C语言中“.”与“->”有什么区别？

从 知乎 转载

转载几篇知乎上我自己的回答，因为不喜欢知乎的排版，所以在博客里重新排版一遍。

原问题：C语言中“.”与“->”有什么区别？

除了表达形式有些不同，功能可以说完全一样阿。那为何又要构造两个功能一样的运算符？效率有差异？可是现在编译器优化都那么强了，如果真是这样岂不是有些多此一举

刚刚翻了下书，说早期的C实现无法用结构直接当作参数在函数间传递，只能用指向结构的指针在函数间进行传递！我想这应该也是最直观的原因吧。

我的回答

首先 `a->b` 的含义是 `(*a).b` ，所以他们是不同的，不过的确 `->` 可以用 `*` 和 `.` 实现，不需要单独一个运算符。 嗯，我这是说现代的标准化的 C 语义上来说， `->` 可以用 `*` 和 `.` 的组合实现。

早期的 C 有一段时间的语义和现代的 C 的语义不太一样。

稍微有点汇编的基础的同学可能知道，在机器码和汇编的角度来看，不存在变量，不存在 struct 这种东西，只存在寄存器和一个叫做内存的大数组。

所以变量，是 C 对内存地址的一个抽象，它代表了一个位置。举个例子，C 里面我们写：

```
1  a = b
```

其实在汇编的角度来看更像是

```
1  *A = *B
```

其中 A 和 B 各是两个内存地址，是指针。

好，以上是基本背景。

基于这个背景我们讨论一下 struct 是什么，以及 struct 的成员是什么。 假设我们有

```
1  struct Point {
2          int x;
3          int y;
4  };
5  struct Point p;
6  struct Point *pp = &p;
```

从现代语义上讲 p 就是一个结构体对象， x 和 y 各是其成员，嗯。

从汇编的语义上讲， p 是一个不完整的地址，或者说，半个地址，再或者说，一个指向的东西是虚构出来的地址。而 x 和 y 各是在 Point 结构中的地址偏移量。也就是说，必须有 p 和 x 或者 p 和 y 同时出现，才形成一个完整的地址，单独的一个 p 没有意义。

早期的 C 就是在这样的模型上建立的。所以对早期的 C 而言， *pp 没有意义，你取得了一个 struct ，而这个 struct 不能塞在任何一个寄存器里，编译器和 CPU 都无法表达这个东西。

这时候只有 p.x 和 p.y 有意义，它们有真实的地址。

早期的 C 就是这样一个看起来怪异的语义，而它更贴近机器的表达。 所以对早期的 C 而言，以下的代码是对的：

```
1  p.x = 1;
2  int *a;
3  a = &(p.x);
```

而以下代码是错的：

```
1  (*pp).x = 1;
```

因为作为这个赋值的目标地址表达式的一部分， *pp ，这个中间结果没法直译到机器码。

所以对早期的 C 而言，对 pp 解引用的操作，必须和取成员的偏移的操作，这两者紧密结合起来变成一个单独的操作，其结果才有意义。

所以早期的 C 就发明了 -> ，表示这两个操作紧密结合的操作。于是才能写：

```
1  pp->x = 1;
```

嗯，这就是它存在的历史原因。 而这个历史原因现在已经不重要了，现代的符合标准的 C 编译器都知道 `(*pp).x` 和 `pp->x` 是等价的了。

说句题外话， C++ 里面还发明了 `.*` 和 `->*` 这两个运算符（注意 `->*` 不是单独的 `->` 和 `*` 并排放的意思），关于为什么要发明这两个运算符，而不能直接说 `a ->* b` 的意思就是 `a ->(*b)` ，这个就作为课堂作业吧。