Use Rust To Make A TSDB

rust入门基础（十六）

Lecturer: ZuoTiJia

Date: 2022.10.26

类型的大小

我们知道，在计算机上，数据不过是01的bit串，而我们人类赋予bit串不同的含义，

00110001

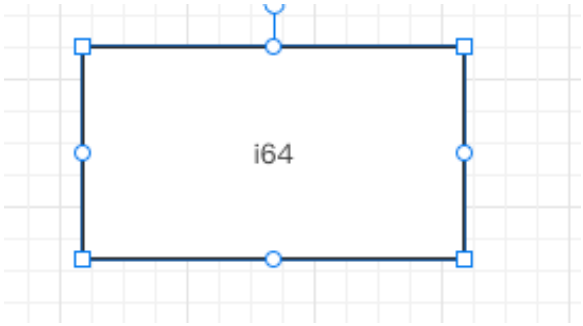
这一个字节的二进制，用ascii码去解释，就是字符 '1'，如果用10进制整数去解释就是49，

数据还是那个数据，但我们人为的去标记识别他。

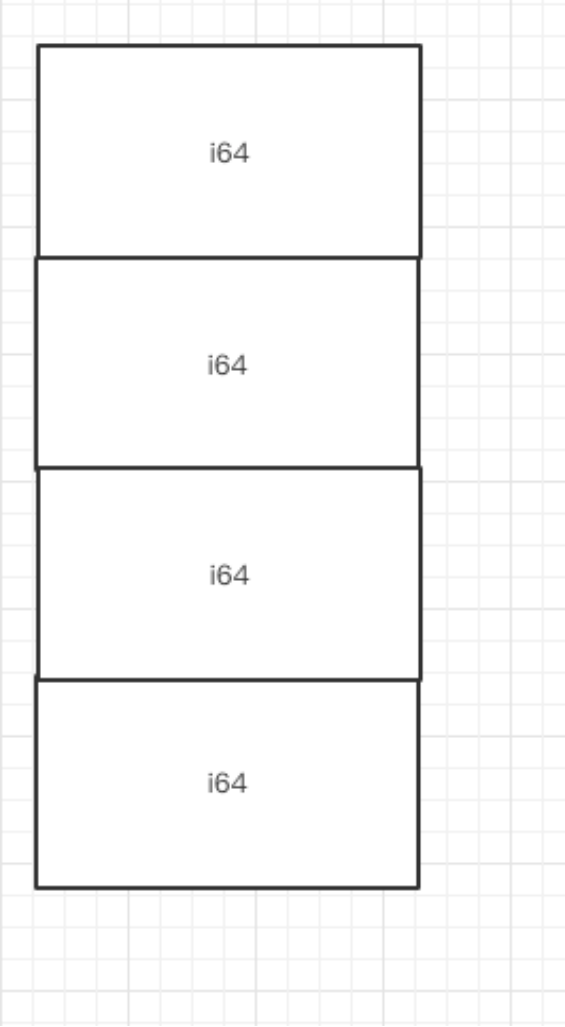
同样一个8字节数据，你可以把它解释成i64，也可以解释成f64。

rust编译器，会知道数据怎么解释成类型的值。而想要解释数据，就得知道，这个类型的数据大小和布局。

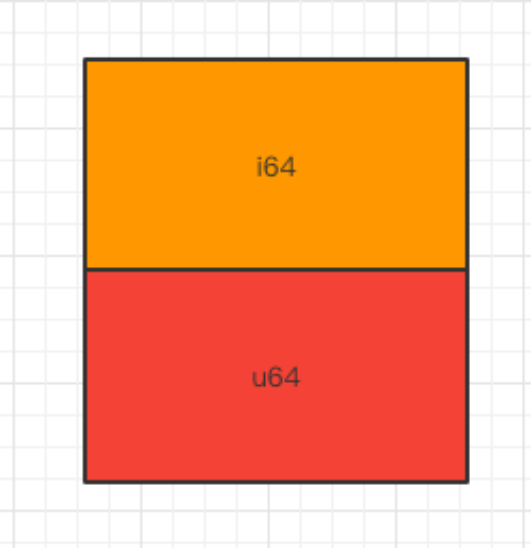
i32, i64 大小



数组的内存布局



struct 的内存布局

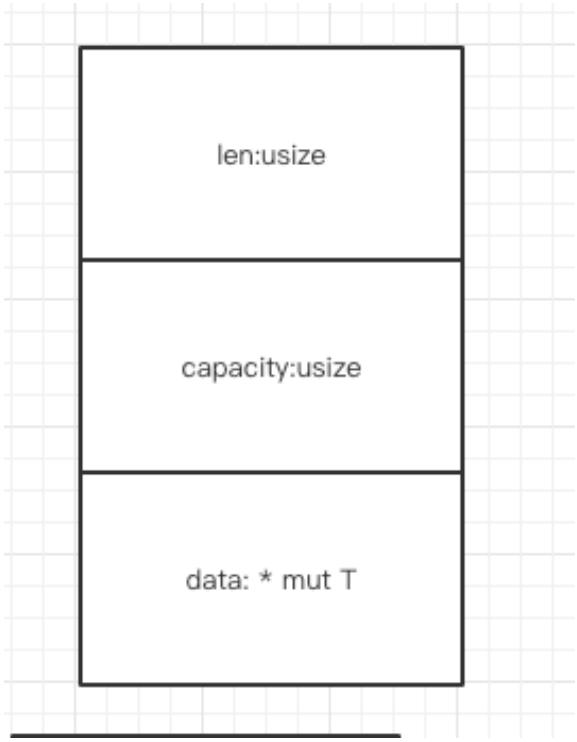


内存对齐

变量顺序重排

Vec的内存布局示意图

```
struct SimpleVec {
    len: usize,
    capacity: usize,
    data: * const T
}
```



Option的内存布局

enum的内存布局

```
enum Data {
    I32(i32),
    F64(f64),
    Bytes(Vec)
```

}

嵌套enum的内存布局

Option

struct Nothing;

type Void = ();

Array0 = [u8;0]

大小为0的类型

struct ZST {

  nothing: Nothing,

  void_: Void,

  array0: Array

}

Zero Sized Type(ZSTs)

Empty Types

enum Empty {}

enum类型的内存布局

rust enum 类似c++ 的std::variant实现，variant大致是用union实现的。

```
fn main() {
  enum Data {
    I32(i32),
    U64(u64),
    Bytes(SimpleVec<u8>)
  }
  println!("sizeof Data:{}", size_of::<Data>());
}
```

enum Data 的内存布局类似下面C++代码

```
//c++ vector 大小也是24
struct Data {
  uint64 tag; //用来标记变体
  union {
```

```
    i64
  }
}
```

```
struct Option {
  uint64 tag; // 占8字节 Some None
  i64;
}
```

如果众多的Data类型变量都只存储8字节的数据，那么空间利用率会大大降低，大量内存未被使用。

标准库Vec的实现

标准库的Vec类型大小是24个字节

```
//实际实现比较复杂，但栈上内存布局与该示意struct一致
struct Vec<T> {
  data: * mut T,  //指向数据的指针
  capacity: usize,//分配内存的大小
  len: usize      //长度
}
```

为了防止多次分配内存影响性能，所以capacity >= len

我们可以看到Vec类型有点大，整整24个字节，在什么地方会影响性能呢?

答案是因为enum类型的内存布局

如果我们改进enum Data的实现，变为如下结构

```
enum Data {
    I64(i64),
    U32(u32),
    U64(u64),
    F64(f64),
    Bytes(Box<Vec<u8>>) // 这里sizeof Box<Vec<u8>> 是8
                        // ! 注意 sizeof Box<[T]> 是16
}
```

可以看到内存占用减少了一半!

但是这种实现的话，有两级指针，使用起来比较蹩脚，而且对系统的缓存也不友好。

creates上有一个库是minivec，解决了这个问题

MiniVec<T> 实现

简而言之，就是把Vec放在栈上的len，capacity移到堆上即可

```
//粗略实现
struct MiniVec<T> {
  data: ptr::Nonptr<u8>
}
impl MiniVec<T> {
  fn head(& mut self ) ->(* mut T, capacity, len) {
    return (data.add(16) as * mut T,
            *data as * const usize,
            *data.add(8) as *const usize)
  }
}
```

Vec<Option<T>> 优化

标准库会为嵌套的enum进行优化

```
enum Data {
  U64(u64),
  F64(f64)
}
//当遇上Option<Data>时，编译器会做展开的优化
//会展开成类似下面实现
enum OptionData {
  SomeU64(u64),
  SomeF64(f64),
  None()
}
//这样 sizeof Data 与 sizeof Option<Data> 大小是一样的
```

但是如果T是一个普通的struct呢

```
struct Data {
  num1: u64,
  num2: i64,
  num3: f64,
}
```

如果我们使用Vec<Option<Data>>

当这个Vec里面None比较多时，空间利用率就大大降低了

creates上有option_vec解决这一麻烦

简略实现如下

```
struct OptionVec<T> {
  vec: Vec<T>,
  bitvec:BitVec
}
impl OptionVec<T> {
  fn push(val: Option<T>) {
    None => {
      vec.push(val);
      bitvec.push(true);
    }
    Some => {
      vec.push(val);
      bitvec.push(false)
    }
  }
  fn get(i: usize) -> Option<T> {
    if bitvec[i] ==
  }
}
```