

Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 12

Коллекции

Выделение Памяти

Выделение Памяти

`std::alloc`

```
impl Layout {  
    pub fn from_size_align(size: usize, align: usize)  
        -> Result<Layout, LayoutErr>;  
    pub fn size(&self) -> usize;  
    pub fn align(&self) -> usize;  
}
```

`std::alloc::Layout` описывает память, которую нужно выделить:

- размер
- ограничения по выравниванию

Для сравнения, `malloc` принимает только размер

Выделение Памяти

`std::alloc`

```
pub unsafe fn alloc(layout: Layout) -> *mut u8  
pub unsafe fn dealloc(ptr: *mut u8, layout: Layout);
```

Аналоги `malloc` и `free`

Safety:

- `layout` должен иметь не нулевой размер,
- `ptr` в `dealloc` должен быть получен из `alloc`



dealloc требует **layout**, можно эффективнее реализовать аллокатор

GlobalAlloc

```
pub unsafe trait GlobalAlloc {  
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;  
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);  
    ...  
}
```

GlobalAlloc — другая сторона API выделения памяти, позволяет поменять аллокаор по умолчанию

Safety:

- трейт **unsafe**, имплементация должна гарантировать, что `alloc` возвращает пустую память
- функции **unsafe** — вызывающий код должен гарантировать, что `dealloc` вызывается корректно

GlobalAlloc

Системный аллокатор (libc::malloc):

lib.rs

```
use std::alloc::System;  
  
#[global_allocator]  
static GLOBAL: System = System;
```

jemalloc:

Cargo.toml

```
[dependencies]  
jemallocator = "0.3.0"
```

lib.rs

```
#[global_allocator]  
static GLOBAL: jemallocator::Jemalloc = jemallocator::Jemalloc;
```

Vec

Box

Почему смотрим на Vec а не на Box?

- Vec — на 100% библиотечный тип, Box — на 90%, как [T]
- Vec — интереснее

Vec

src/liballoc/vec.rs

```
pub struct Vec<T> {  
    buf: RawVec<T>,  
    len: usize,  
}
```

src/liballoc/raw_vec.rs

```
pub struct RawVec<T, A: Alloc = Global> {  
    ptr: Unique<T>,  
    cap: usize,  
    a: A,  
}
```



Разделение **struct** и **impl** это большой бонус к читаемости: 7 строк, и уже понятна имплементация в общих чертах

Vec

src/liballoc/vec.rs

```
pub struct Vec<T> {  
    buf: RawVec<T>,  
    len: usize,  
}
```

src/liballoc/raw_vec.rs

```
pub struct RawVec<T, A: Alloc = Global> {  
    ptr: Unique<T>,  
    cap: usize,  
    a: A,  
}
```

RawVec — управляет памятью под cap объектов, не вызывает деструкторы

Vec — RawVec + len, вызывает drop

RawVec

```
pub struct RawVec<T, A: Alloc = Global> {  
    ptr: Unique<T>, ①  
    cap: usize,  
    a: A,  
}
```

```
pub struct Unique<T: ?Sized> {  
    pointer: *const T, ③  
    _marker: PhantomData<T>, ②  
}
```

- ① указатель на буфер
- ② маркер для dropcheck: Unique<T> владеет T
- ③ ***const** для ковариантности

RawVec

```
pub struct RawVec<T, A: Alloc = Global> {  
    ptr: Unique<T>,  
    cap: usize, ①  
    a: A,  
}
```

① размер буфера (не все элементы инициализированы)

RawVec

```
pub struct RawVec<T, A: Alloc = Global> { ❷  
    ptr: Unique<T>,  
    cap: usize,  
    a: A, ❶  
}
```

❶ аллокатор данного вектора

❷ по умолчанию — Global

Global — ZST, размер вектора с дефолтным аллокатором — 24

Размер RawVec<T, &'a MyBumpAllocator> — 32

```

impl<T> Vec<T> {
    pub const fn new() -> Vec<T> {
        Vec { buf: RawVec::new(), len: 0 }
    }
}

impl<T, A: Alloc> RawVec<T, A> {
    pub const fn new_in(a: A) -> Self {
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        RawVec {
            ptr: Unique::empty(),
            cap,
            a,
        }
    }
}

```

- new не аллоцирует — полезный паттерн
- ZST не магия — нужна явная поддержка со стороны stdlib

Deref / DerefMut

```
impl<T> ops::Deref for Vec<T> {  
    type Target = [T];  
  
    fn deref(&self) -> &[T] {  
        unsafe {  
            let p = self.buf.ptr();  
            assume(!p.is_null()); 1  
            slice::from_raw_parts(p, self.len)  
        }  
    }  
}
```

1 указание оптимизатору

Бесплатно получили все методы `&[T]` и `&mut [T]`, в том числе итераторы по ссылкам

push

```
impl<T> for Vec<T> {  
    pub fn push(&mut self, value: T) {  
        if self.len == self.buf.cap() { self.reserve(1); } ❶  
        unsafe {  
            let end = self.as_mut_ptr().add(self.len); ❷  
            ptr::write(end, value); ❸  
            self.len += 1;  
        } ❹  
    }  
}
```

❶ самое интересное, удвоение размера, тут

❷ `as_mut_ptr` — метод на `&mut [T]`

❸ `unsafe fn write<T>(dst: *mut T, src: T) — memcpy + mem::forget`

❹ инвариант - первые `len` элементов инициализированы

set_len

```
impl<T> Vec<T> {  
    pub unsafe fn set_len(&mut self, new_len: usize) {  
        debug_assert!(new_len <= self.capacity());  
  
        self.len = new_len;  
    }  
}
```

set_len — публичный **unsafe** метод, иногда позволяет заполнить вектор эффективнее, чем push или collect

Метод не содержит **unsafe** операций, если убрать **unsafe**, код *будет* компилироваться

unsafe и инварианты



unsafe блок заражает весь модуль: ошибка снаружи
unsafe может нарушить memory safety

При корректном использовании **unsafe** всё публичное API должно безопасно

```
// OK!
```

```
pub unsafe fn set_len(&mut self, new_len: usize)
```

```
// Так себе, но формально OK
```

```
fn set_len(&mut self, new_len: usize)
```

```
// Неправильный unsafe код в Deref
```

```
pub fn set_len(&mut self, new_len: usize)
```

pop

```
impl <T> Vec<T> {  
    pub fn pop(&mut self) -> Option<T> {  
        if self.len == 0 {  
            None  
        } else {  
            unsafe {  
                self.len -= 1;  
                Some(ptr::read(self.get_unchecked(self.len())))  
            }  
        }  
    }  
}
```

- `get_unchecked` — метод `&[T]`
- **unsafe fn** `read<T>(src: *const T) -> T` — антоним `write`
- память при `pop` не освобождается (есть `shrink_to_fit`)

Drop

```
unsafe impl<#[may_dangle] T> Drop for Vec<T> { ❶  
    fn drop(&mut self) {  
        unsafe {  
            ptr::drop_in_place(&mut self[..]); ❷  
        }  
    } ❸  
}
```

- ❶ dropcheck eyepatch: Vec<&'a T> может чуть-чуть пережить 'a
- ❷ ручной вызов деструкторов для первых len элементов
- ❸ вызов drop у buf: RawVec<T>, который освобождает память

reserve

```
impl<T> Vec<T> {  
    pub fn reserve(&mut self, additional: usize) {  
        self.buf.reserve(self.len, additional); ❶  
    }  
}
```

- ❶ передаём в RawVec текущую длину: при выделении нового буфера, RawVec скопирует первые `len` элементов

```

impl<T> RawVec<T> {
    pub fn reserve(
        &mut self,
        used_cap: usize,
        needed_extra_cap: usize,
    ) {
        let res = self.reserve_internal(
            used_cap, needed_extra_cap, Infallible, Amortized
        );
        match res {
            Err(CapacityOverflow) => capacity_overflow(),
            Err(AllocErr) => unreachable!(),
            Ok(()) => { /* yay */ }
        }
    }
}

```

Просим аллоцировать в два раза больше (Amortized)

В случае ошибки аллокатора — abort (Infallible)

```

fn amortized_new_size(
    &self,
    used_cap: usize,
    needed_extra_cap: usize,
) -> Result<usize, CollectionAllocErr> {
    // Nothing we can really do about these checks :(
    let required_cap = used_cap.checked_add(needed_extra_cap)
        .ok_or(CapacityOverflow)?;
    // Cannot overflow, because `cap <= isize::MAX`,
    // and type of `cap` is `usize`.
    let double_cap = self.cap * 2;
    // `double_cap` guarantees exponential growth.
    Ok(cmp::max(double_cap, required_cap))
}

```

Коэффициент роста — 2, есть гипотеза, что 1.5 — лучше:

github.com/facebook/folly/.../FBVector.md#memory-handling


```
fn reserve_internal(  
    &mut self,  
    used_cap: usize,  
    needed_extra_cap: usize,  
    fallibility: Fallibility,  
    strategy: ReserveStrategy,  
) -> Result<(), CollectionAllocErr> {  
    unsafe {  
        ...  
    }  
}
```

```

fn reserve_internal( ... ) -> Result<(), CollectionAllocErr> {
    unsafe {
        if self.cap().wrapping_sub(used_cap) >= needed_extra_cap {
            return Ok(()); ❶
        }
        let new_cap = match strategy { ❷
            Exact => used_cap.checked_add(needed_extra_cap)
                .ok_or(CapacityOverflow)?,
            Amortized => self.amortized_new_size(
                used_cap,
                needed_extra_cap,
            )?,
        };
        ...
    }
}

```

❶ fast path: место уже есть (ZST попадаю сюда)

❷ посчитали новую capacity:

`max(self.cap * 2, used_cap + needed_extra_cap)`

```
fn reserve_internal( ... ) -> Result<(), CollectionAllocErr> {  
    unsafe {  
        ...  
  
        let new_layout = Layout::array::<T>(new_cap)  
            .map_err(|_| CapacityOverflow)?; ❶  
  
        alloc_guard(new_layout.size())?; ❷  
  
        ...  
    }  
}
```

- ❶ посчитали size и alignment для массива из new_cap элементов типа T
- ❷ проверили технический инвариант: size <= isize::MAX

```

fn reserve_internal( ... ) -> Result<(), CollectionAllocErr> {
    unsafe {
        ...
        let res = match self.current_layout() { ❶
            Some(layout) => {
                self.a.realloc(
                    NonNull::from(self.ptr).cast(),
                    layout, new_layout.size(),
                ) ❸
            }
            None => self.a.alloc(new_layout), ❷
        };
        ...
    }
}

```

- ❶ посчитали текущий Layout (None если cap == 0)
- ❷ случай cap == 0, просто аллоцируем память
- ❸ магия!

realloc

```
pub unsafe trait GlobalAlloc {  
    unsafe fn realloc(  
        &self,  
        ptr: *mut u8,  
        layout: Layout,  
        new_size: usize  
    ) -> *mut u8 { ... }  
}
```

`realloc` выделяет новую память, копирует данные, освобождает старую память

В Rust, `realloc` работает для *любых* объектов, так как `move` есть всегда и это memcpy

В C++ будут вызваны **move** конструкторы, если они поехсерт

realloc

Хорошие аллокаторы не копируют данные при `realloc`, а меняют таблицу страниц (`mmap`)

`realloc` это магия!

```

fn reserve_internal( ... ) -> Result<(), CollectionAllocErr> {
    unsafe {
        ...
        let res = match self.current_layout() {
            Some(layout) => {
                self.a.realloc(
                    NonNull::from(self.ptr).cast(),
                    layout, new_layout.size(),
                )
            }
            None => self.a.alloc(new_layout),
        };
        ...
    }
}

```

```

fn reserve_internal( ... ) -> Result<(), CollectionAllocErr> {
    unsafe {
        ...
        match (&res, fallibility) { ❶
            (Err(AllocErr), Infallible) =>
                handle_alloc_error(new_layout), // -> !
            _ => {}
        }

        self.ptr = res?.cast().into(); ❶ ❷
        self.cap = new_cap; ❸

        Ok(())
    }
}

```

- ❶ обработали ошибку
- ❷ превратили `*mut u8` в `*mut T`
- ❸ записали новую capacity


```

unsafe impl<#[may_dangle] T, A: Alloc> Drop for RawVec<T, A> {
    fn drop(&mut self) {
        unsafe { self.dealloc_buffer(); }
    }
}

```

```

impl<T, A: Alloc> RawVec<T, A> {
    /// Frees the memory owned by the RawVec *without*
    /// trying to Drop its contents.
    pub unsafe fn dealloc_buffer(&mut self) {
        let elem_size = mem::size_of::<T>();
        if elem_size != 0 {
            if let Some(layout) = self.current_layout() {
                self.a.dealloc(
                    NonNull::from(self.ptr).cast(),
                    layout,
                );
            }
        }
    }
}

```

Vec

- RawVec управляет куском памяти, Vec управляет уничтожением объектов
- рост в два раза
- реализация полагается на то, что move = memcpy
- весь код потенциально **unsafe**, потому что может нарушить инвариант len

Intolter

```
impl<T> IntoIterator for Vec<T> {  
    type Item = T;  
    type IntoIter = IntoIter<T>;  
  
    /// Creates a consuming iterator.  
    fn into_iter(mut self) -> IntoIter<T> { ... }  
}  
  
pub struct IntoIter<T> {  
    buf: NonNull<T>,  
    phantom: PhantomData<T>,  
    cap: usize,  
    ptr: *const T,  
    end: *const T,  
}
```

```

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        unsafe {
            if self.ptr as *const _ == self.end {
                return None
            }

            // ZST -- особый случай
            if mem::size_of::<T>() == 0 {
                self.ptr = arith_offset(
                    self.ptr as *const i8, 1
                ) as *mut T;
                return Some(mem::zeroed())
            }
            let old = self.ptr;
            self.ptr = self.ptr.offset(1);
            Some(ptr::read(old))
        }
    }
}

```

Intolter

```
unsafe impl<#[may_dangle] T> Drop for IntoIter<T> {  
    fn drop(&mut self) {  
        // destroy the remaining elements  
        for _x in self.by_ref() {}  
  
        // RawVec handles deallocation  
        let _ = unsafe {  
            RawVec::from_raw_parts(self.buf.as_ptr(), self.cap)  
        };  
    }  
}
```

Drain

```
let before = xs.len();  
for x in xs.drain(10..20) {  
    process(x)  
}  
assert_eq!(xs.len(), before - 10);
```

`drain` это consuming итератор по части элементов

В `drop Drain` сдвигает элементы в образовавшуюся дырку

Композиция unsafe

Leak Amplification

Нет гарантии, что `drop` будет вызван

В конструкторе `Drain` устанавливает длину вектора в 0, в деструкторе — восстанавливает

Если утечь `Drain`, то не будут вызваны деструкторы оставшихся элементов вектора, leak amplification



Если бы **`Drain`** не обнулял длину, то можно было бы получить доступ к не валидным элементам из дырки!

Leakpocalypse Now!

В Rust 1.0 функция `mem::forget` была помечена **unsafe**

Но можно написать безопасный `forget`!

Leakpocalypse Now!

Трюк — можно создать цикл из счётчиков ссылок

Хватит цикла длины 1:

```
fn safe_forget<T>(value: T) {  
    use std::cell::RefCell;  
    use std::rc::Rc;  
  
    struct Holder<T> {  
        value: T,  
        link: RefCell<Option<Rc<Holder<T>>>>,  
    }  
  
    let holder = Holder { value, link: RefCell::new(None) };  
    let holder = Rc::new(holder);  
    let holder2 = Rc::clone(&holder);  
    *holder.link.borrow_mut() = Some(holder2);  
}
```

Leakpocalypse Now!

В Rust, где `mem::forget` это **unsafe** функция можно:

1. Написать `Rc<T>`, и это будет ОК.
2. Написать `thread::scoped`, полагающийся на вызов деструктора, и это тоже будет ОК!

Но 1 + 2 вместе ведут к `use after free`!

По отдельности, **unsafe** блоки в `Rc` и в `thread::scoped` корректны, но вместе кто-то из них не прав

[observational-equivalence-and-unsafe-code](#)

`std::collections`

Кэш

Latency Comparison Numbers (~2012)

| | | | |
|-----------------------|-----|----|---------------|
| ----- | | | |
| L1 cache reference | 0.5 | ns | |
| Branch mispredict | 5 | ns | |
| L2 cache reference | 7 | ns | |
| Mutex lock/unlock | 25 | ns | |
| Main memory reference | 100 | ns | 200x L1 cache |

[interactive latency.html](http://interactive.latency.html)

CPU существенно быстрее памяти

Ключ к производительности — компактные структуры данных с хорошей spatial locality

Массив лучше связного списка

BTreeMap

Сбалансированное дерево поиска (K: Ord)

В узлах хранится [(K, V); CAPACITY]

```
const B: usize = 6;  
pub const MIN_LEN: usize = B - 1;  
pub const CAPACITY: usize = 2 * B - 1;
```

Быстрее, чем красно-чёрное или AVL дерево

В C++ `std::map` нельзя реализовать при помощи B-дерева из-за гарантий инвалидации итераторов

BTreeSet

```
pub struct BTreeSet<T> {  
    map: BTreeMap<T, ()>,  
}
```

() это ZST: ничего лишнего не храним!

HashMap / HashSet

Для хэш таблиц есть две основные стратегии реализации:

- chaining — bucket элементов с совпадающим хэшем это `LinkedList`
- open addressing — все элементы хранятся в одном массиве, при коллизии записываем элемент в соседний слот

В Rust — SIMD accelerated open addressing (state of the art!):

<https://github.com/rust-lang/hashbrown>

В C++ API `std::unordered_map` открывает chaining как деталь реализации

Хэш Функция

HashMap / HashSet параметризованы хэш функцией

По умолчанию — SipHash: не самый быстрый, но очень collision resistant хэш

FxHash — подвержен DoS, но быстрее, используется в компиляторе

Entry API

```
if !map.contains(key) {  
    map.insert(key, value)  
}
```

Performance code smell: ключ key ищется в коллекции два раза

```
use std::collections::hash_map::{HashMap, Entry};  
  
match counts.entry(key) {  
    Entry::Vacant(entry) => { entry.insert(1); },  
    Entry::Occupied(entry) => *entry.into_mut() += 1,  
}  
  
*counts.entry(key).or_insert_with(|| 0) += 1;
```



API, похожее на `entry`, есть во многих языках программирования!

Ещё Коллекции

- VecDeque — очередь, ring buffer внутри
- BinaryHeap — очередь с приоритетом (интересная **unsafe** реализация sift-up, минимизирующая копирования)
- LinkedList

https://github.com/gnzlbgr/slice_deque:

безумные трюки с виртуальной памятью;

VecDeque, на который можно посмотреть как на [T]

<https://github.com/BurntSushi/fst>:

Хранение множества строк в виде автомата, хороший способ написать fuzzy search