How NOT to Rust

喵喵

2021.10

How NOT to Rust

本来想用的标题是 A Brief, Incomplete, and Mostly Wrong History of Rust[1]。

[1] http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

How NOT to Rust

本来想用的标题是 A Brief, Incomplete, and Mostly Wrong History of Rust[1]。

但是喵喵写 Slide 毫无灵感...

图: 呼呼喵喵

[1] http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

How NOT to Rust

本来想用的标题是 A Brief, Incomplete, and Mostly Wrong History of Rust[1]。

但是喵喵写 Slide 毫无灵感...

图: 精神喵喵!!

[1] http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

How NOT to Rust

来聊聊 Rust 的历史。(其实大多数是喵喵的 Rant)

How NOT to Rust

来聊聊 Rust 的历史。(其实大多数是喵喵的 Rant)
这个 Talk 包括:

- Rust Overview!
- Rust 的各种 Joke

这个 Talk 不包括:

- Rust 详细入门，请查看 The Rust Programming Language
- Rust 详细语义，请查看 The Rust Language Reference
- 讲者存在 Rust 深刻知识的任何可能。

How NOT to Rust

来聊聊 Rust 的历史。(其实大多数是喵喵的 Rant)
这个 Talk 包括:

- Rust Overview!
- Rust 的各种 Joke

这个 Talk 不包括:

- Rust 详细入门，请查看 The Rust Programming Language
- Rust 详细语义，请查看 The Rust Language Reference
- 讲者存在 Rust 深刻知识的任何可能。

喵喵刚刚入门 Rust，请爱护喵喵!

快速 Rust 入门

Rust 和 C(|++), Java(|Script), Python, LISP, … 有什么不同？

快速 Rust 入门

Rust 和 `C(|++)`, `Java(|Script)`, `Python`, `LISP`, … 有什么不同？

- Algebraic Data Types
- `crate` and `module`
- Polymorphism: Generic + Trait
- (The dreaded) Lifetime / `borrowck`

ADT: Algebraic Data Types

```
// KV DB, Client -> Server conn payload
enum KVPayload {
  Close,
  Get(String),
  Put {
    key: String,
    value: String,
    expire: Datetime,
  },
}
```

ADT: Algebraic Data Types

```
// KV DB, Client -> Server conn payload
enum KVPayload {
  Close,
  Get(String),
  Put {
    key: String,
    value: String,
    expire: Datetime,
  },
}
```

Disjoint sum over products!

Generic over type

```
enum SingleOrVec<T> {
  Single(T),
  Vec(Vec<T>),
}
```

Generic over type...?

Heap allocation are bad.

Generic over type...?

Heap allocation are bad.

```
enum SingleOrArray<T, const N: usize> {
  Single(T),
  Array([T; N]),
}
```

Generic over type...?

Heap allocation are bad.

```
enum SingleOrArray<T, const N: usize> {
  Single(T),
  Array([T; N]),
}
```

Const generics (rfcs#2000)

Module-level encapsulation

Crate = Node/Go/Python package (sort of)
功能集合，例如：`serde` 提供了序列化、反序列化相关的基础设施。

Module-level encapsulation

Crate = Node/Go/Python package (sort of)
功能集合，例如：`serde` 提供了序列化、反序列化相关的基础设施。

Module = Java package (sort of)
实现单元，“可见性“的边界。例如：`serde::ser` 包含序列化 (Serialize) 相关的声明和实现。

Module-level encapsulation

Crate = Node/Go/Python package (sort of)
功能集合，例如：`serde` 提供了序列化、反序列化相关的基础设施。

Module = Java package (sort of)
实现单元，“可见性“的边界。例如：`serde::ser` 包含序列化 (Serialize) 相关的声明和实现。

```
mod data;
pub use data::{Input, Output};
```

Abusing modules

library/std/src/os/mod.rs:

```
// unix
#[cfg(not(all(
    doc,
    any(
        all(target_arch = "wasm32", not(target_os =
↪   "wasi")),
        all(target_vendor = "fortanix", target_env =
↪   "sgx")
    )
)))]
#[cfg(target_os = "hermit")]
#[path = "hermit/mod.rs"]
pub mod unix;
```

C style preprocessor?

C style preprocessor?

With procedure macro?

```
preprocess!{
  #ifdef PLATFORM_WINDOWS
    // Do something
  #else
    // Do something else
  #endif
}
```

C style preprocessor?

~~With procedure macro?~~

```
preprocess!{
  #ifdef PLATFORM_WINDOWS
    // Do something
  #else
    // Do something else
  #endif
}
```

Please don't.

Polymorphism

- Composition over Inheritance 设计模式!
- Trait: 描述一个接口

Polymorphism

- Composition over Inheritance 设计模式!
- Trait: 描述一个接口
- Generic: 使用 Trait 的静态派发
- Trait Objects: 使用 Trait 的动态派发

Polymorphism, Cont.

```
trait Monoid {
  // "Member functions"
  fn product(&self, other: &Self) -> Self;
  // "Static functions"
  fn identity() -> Self;
  fn is_commutative() -> bool;
}

trait Group: Monoid {
  fn inverse(&self) -> Self;
  // Default impl
  fn is_abelian() -> bool {
    return Self::is_commutative();
  }
}
```

Polymorphism, Cont.

```
struct Cyclic<const N: usize>(usize);

impl<const N: usize> Monoid for Cyclic<N> {
    fn product(&self, ano: &Self) -> Self {
        let inner = self.0;
        let anoinner = ano.0;
        return Self((self.0 + ano.0) % N)
    }
    fn identity() -> Self { Self(0) }
    fn is_commutative() -> bool { true }
}
```

Polymorphism, Cont.

```
fn subgroup_by<G>(gen: G) -> Vec<G>
  where G: Group + Eq + Clone
{
  let mut cur = gen.clone();
  let mut result = Vec::new();
  loop {
    result.push(cur.clone());
    cur = cur.product(gen);
    if cur == gen {
      return result;
    }
  }
}
```

Polymorphism, Cont.

```
let trait_obj: &dyn Group = &group;
```

Polymorphism, Cont.

```
let trait_obj: &dyn Group = &group;
```

VTable with fat pointer

Polymorphism, Cont.

```
let trait_obj: &dyn Group = &group;
```

VTable with fat pointer

Polymorphism, Cont.

```
let trait_obj: &dyn Group = &group;
```

VTable with fat pointer

```
let boxed_fn: Box<dyn Fn(usize) -> usize> =
  Box::new(
      |input: usize| -> usize {
          input * 2
      }
  );
```

Polymorphism... ?

```
trait Trait {}
fn generic_fn<T: Trait + ?Sized>() {
  println!("{}", size_of::<&dyn Trait>()); // -> 16
  println!("{}", size_of::<&T>());         // -> 8
}
```

Polymorphism... ?

```
trait Trait {}
fn generic_fn<T: Trait + ?Sized>() {
  println!("{}", size_of::<&dyn Trait>()); // -> 16
  println!("{}", size_of::<&T>());         // -> 16
}

fn main() {
  generic_fn::<dyn Trait>();
}
```

Why Traits?

Why Traits?

没有“基类对象”，没有“菱形继承”。

Why Traits?

没有“基类对象”，~~没有“菱形继承”~~。

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB> Common for T { }
```

Why Traits?

没有“基类对象”，~~没有“菱形继承”~~。

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB> Common for T { }
```

Rust ~~(尝试)~~ 禁止这件事情。

- Orphan rule

Why Traits?

没有“基类对象”，~~没有“菱形继承”~~。

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB> Common for T { }
```

Rust ~~(尝试)~~ 禁止这件事情。

- Orphan rule
- "Strictly more specified"

Why Traits (More Solid)

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB + SpecA> Common for T { }
```

Why Traits (More Solid)

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB + SpecA> Common for T { }
```

You'll need

```
#![feature(specialization)]
```

See also: RFC 1210: Specialization

Why Traits (More Solid)

```
trait Common {}
trait SpecA {}
trait SpecB {}

impl<T: SpecA> Common for T { }
impl<T: SpecB + SpecA> Common for T { }
```

You'll need

```
#![feature(specialization)]
```

See also: RFC 1210: Specialization

"Chalk"

impl Trait for (A, B, C)

```
[-] impl<A, B, C, D, E, F, G, H, I, J, K, L> PartialOrd<(A, B, C, D, E, F, G, H, I, J, K, L)> for  [src]
    (A, B, C, D, E, F, G, H, I, J, K, L)
    where
        C: PartialOrd<C> + PartialEq<C>,
        F: PartialOrd<F> + PartialEq<F>,
        K: PartialOrd<K> + PartialEq<K>,
        I: PartialOrd<I> + PartialEq<I>,
        E: PartialOrd<E> + PartialEq<E>,
        H: PartialOrd<H> + PartialEq<H>,
        B: PartialOrd<B> + PartialEq<B>,
        A: PartialOrd<A> + PartialEq<A>,
        J: PartialOrd<J> + PartialEq<J>,
        G: PartialOrd<G> + PartialEq<G>,
        D: PartialOrd<D> + PartialEq<D>,
        L: PartialOrd<L> + PartialEq<L> + ?Sized,
```

What about array?

[-] `impl<T, const N: usize> AsMut<[T]> for [T; N]` [src]

[-] `pub fn as_mut(&mut self) -> &mut [T]` ⓘ [src]

Performs the conversion.

[-] `impl<T, const N: usize> AsRef<[T]> for [T; N]` [src]

[-] `pub fn as_ref(&self) -> &[T]` ⓘ [src]

Performs the conversion.

[-] `impl<T, const N: usize> Borrow<[T]> for [T; N]` 1.4.0 [src]

[-] `pub fn borrow(&self) -> &[T]` ⓘ [src]

Immutably borrows from an owned value. Read more

[-] `impl<T, const N: usize> BorrowMut<[T]> for [T; N]` 1.4.0 [src]

[-] `pub fn borrow_mut(&mut self) -> &mut [T]` ⓘ [src]

Mutably borrows from an owned value. Read more

[-] `impl<T, const N: usize> Debug for [T; N]` [src]

```
where
    T: Debug,
```

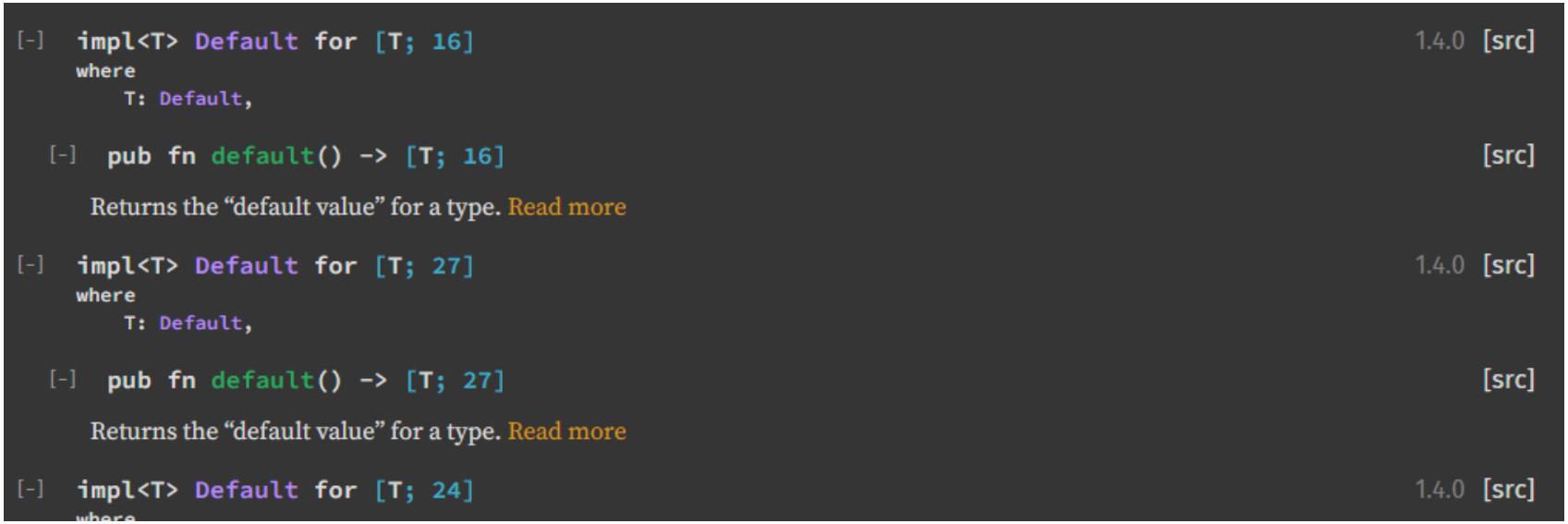
[-] `pub fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>` [src]

Formats the value using the given formatter. Read more

What about array....???

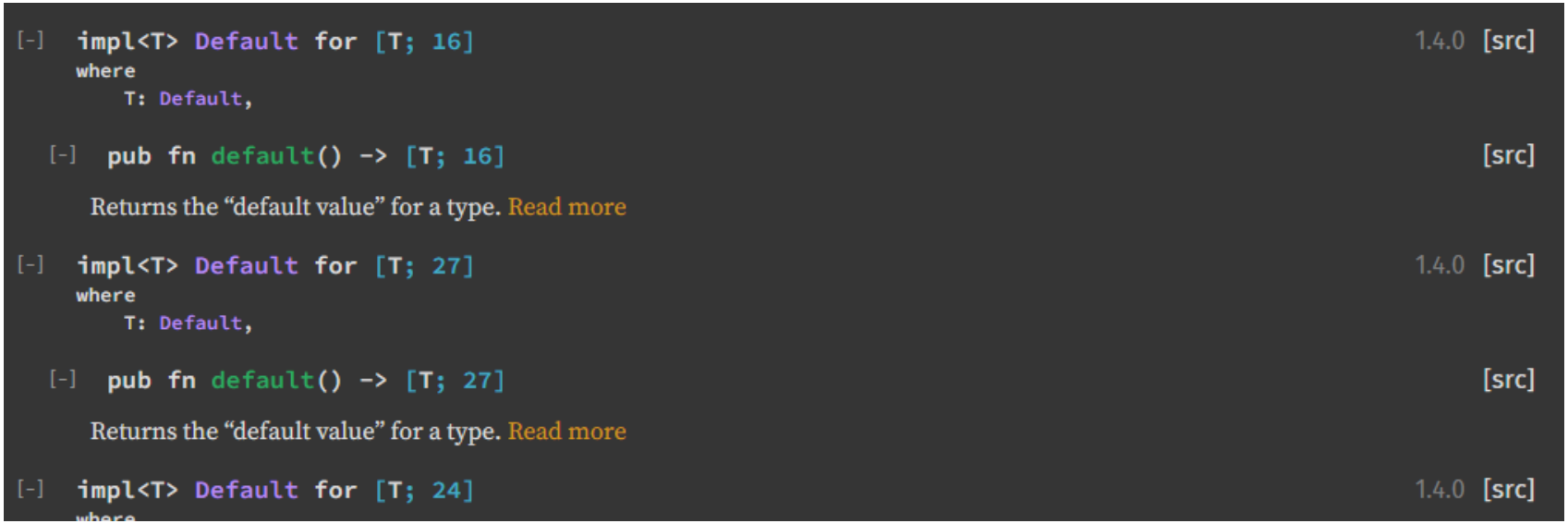
```
[-] impl<T> Default for [T; 16]                                    1.4.0 [src]
    where
        T: Default,

  [-] pub fn default() -> [T; 16]                                        [src]

      Returns the "default value" for a type. Read more

[-] impl<T> Default for [T; 27]                                    1.4.0 [src]
    where
        T: Default,

  [-] pub fn default() -> [T; 27]                                        [src]

      Returns the "default value" for a type. Read more

[-] impl<T> Default for [T; 24]                                    1.4.0 [src]
    where
```

What about array....???




```
impl<T: const N: usize> Default for [T; N]
  where T: Default;
```

What about array....???




```
impl<T: const N: usize> Default for [T; N]
  where T: Default;
impl<T> Default for [T; 0];
```

Trait: Operator Overloading

Trait: Operator Overloading

```
struct Point(f64, f64);
impl Add for Point {
  type Output = Point;
  fn add(self, rhs: Point) -> Point {
    let Point(sx, sy) = self;
    let Point(rs, ry) = rhs;
    Point(sx + rs, sy + ry)
  }
}
```

Trait: Operator Overloading, Cont.

```
#[lang = "add"]
pub trait Add<Rhs = Self> {
    // ...
}
```

Trait: Operator Overloading, Cont.

```rust
#[lang = "add"]
pub trait Add<Rhs = Self> {
    // ...
}

impl Add for usize {
    type Output = usize;

    #[inline]
    #[rustc_inherit_overflow_checks]
    fn add(self, rhs: usize) -> usize {
        self + rhs
    }
}
```

Let's look at Box

```
impl<T: ?Sized, A: Allocator> Deref for Box<T, A> {
  type Target = T;

  fn deref(&self) -> &T {
      &**self
  }
}
```

Let's look at Box

```
impl<T: ?Sized, A: Allocator> Deref for Box<T, A> {
  type Target = T;

  fn deref(&self) -> &T {
      &**self
  }
}
```

Box is a "primitive"

But wait...

```
struct MeowBox<T> {
  ptr: *mut T,
}

impl<T> MeowBox<T> {
  fn new(e: T) -> MeowBox<T> {
    unsafe {
      let space = alloc();
      ptr::write(space, e);
    }
    MeowBox<T> {
      ptr: space
    }
  }
}
```

Box is special

```
let old_school: ~usize = ~10;
let now: Box<usize> = Box::new(10);
let now_really: Box<usize> = box 10;
```

Box is special

```
let old_school: ~usize = ~10;
let now: Box<usize> = Box::new(10);
let now_really: Box<usize> = box 10;
```

你可以从 Box 中把东西拿出来: DerefMove(rfcs#997)

```
struct NoClone(usize);
let boxed = Box::new(NoClone(0));
let inner = *boxed; // Box is invalid now
```

But there's a cost...

```
let large = Box::new([0; 1000000]);
```

But there's a cost...

```
let large = Box::new([0; 1000000]);
```

Guaranteed Copy Elision? Not yet.

Detour: Optimization

```
fn main() {
  (|| (loop {}))()
}
```

Detour: Optimization

```
fn main() {
  (|| (loop {}))()
}
```

Illegal Instruction (#28728)
"Forward progress guarantee"

Detour: Optimization

```
fn main() {
  (|| (loop {}))()
}
```

Illegal Instruction (#28728)
"Forward progress guarantee"
LLVM 12 & cranelift & gccrs

"But that's LLVM's fault!"

```
enum Option<T> {
  None, Some(T),
}
size_of::<Option<bool>>();
size_of::<Option<&T>>();
```

"But that's LLVM's fault!"

```
enum Option<T> {
  None, Some(T),
}

size_of::<Option<bool>>(); // 1
size_of::<Option<&T>>();   // 8
```

"But that's LLVM's fault!"

```
enum Option<T> {
  None, Some(T),
}
size_of::<Option<bool>>(); // 1
size_of::<Option<&T>>();   // 8
size_of::<Option<MaybeUninit<bool>>>(); // 2
size_of::<Option<MaybeUninit<&T>>>();   // 16
```

Borrow checker

核心目标：

- 读不了非法内存 (Uninitialized, Use after freed)
- 比较难 Race (一段代码、一个线程在读，另外一段代码、一个线程在写)

Borrow checker

核心目标：

- 读不了非法内存 (Uninitialized, Use after freed)
- 比较难 Race (一段代码、一个线程在读，另外一段代码、一个线程在写)

Lifetime!

Lifetime

```
let mut slot: Option<&usize> = None;
{
  let data = 10usize;
  slot = Some(&data); // Error!
}
println!("{}", slot.unwrap());
```

Lifetime, Cont.

```
fn main() {
  let on_stack = 10usize;
  thread::spawn(|| { // Error!
    println!("{}", on_stack);
  });
}
```

But if we really need...

But if we really need...

Thread guards:

```
fn main() {
  let on_stack = 10usize;
  let guard = thread::scoped(|| {
    println!("{}", on_stack);
  });

  // guard impls Drop (dtor)
  // Thread joins here
}
```

Oops

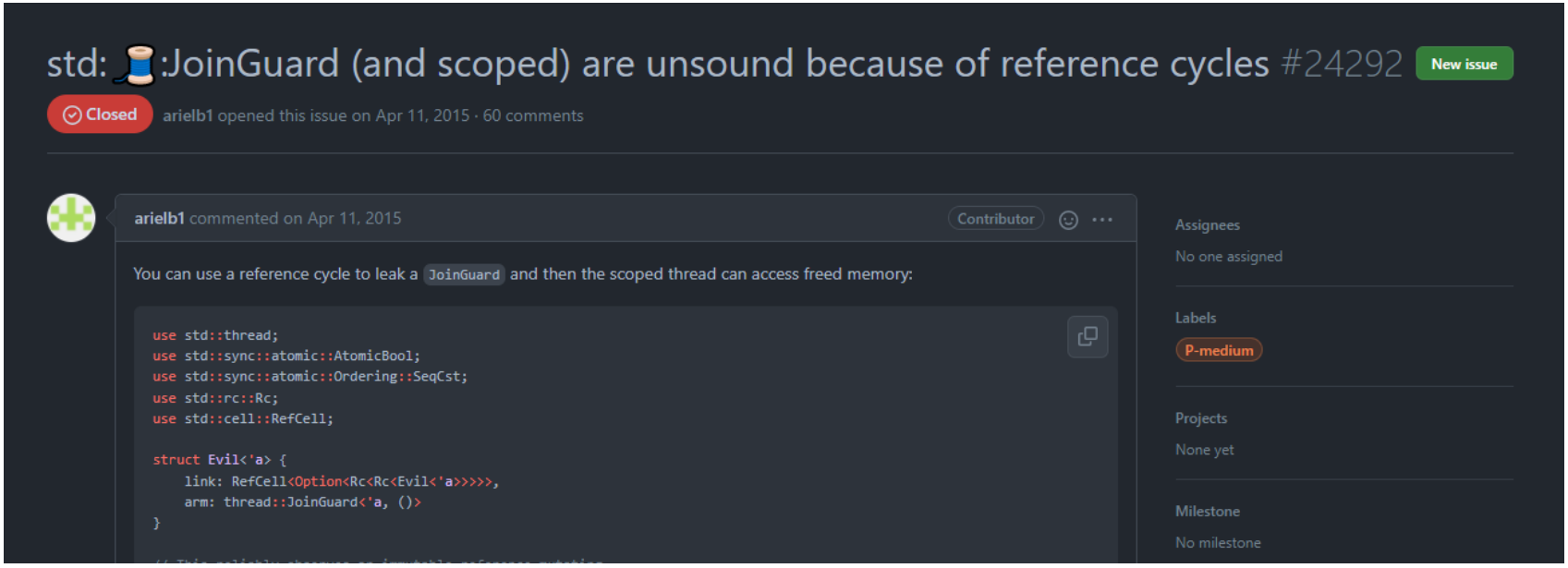


图:

```
struct Evil<'a> {
  link: RefCell<Option<Rc<Rc<Evil<'a>>>>>,
  arm: thread::JoinGuard<'a, ()>
}
```

rfcs#1066

`mem::forget` 去掉了 `unsafe`.

rfcs#1066

`mem::forget` 去掉了 `unsafe`.

- `Drain` 需要确保只移出了一部分时不会访问非法内存。
- `Arc` 需要考虑溢出。
- `thread::scoped` 被完全删除了。

Last but not least...

https://turbo.fish

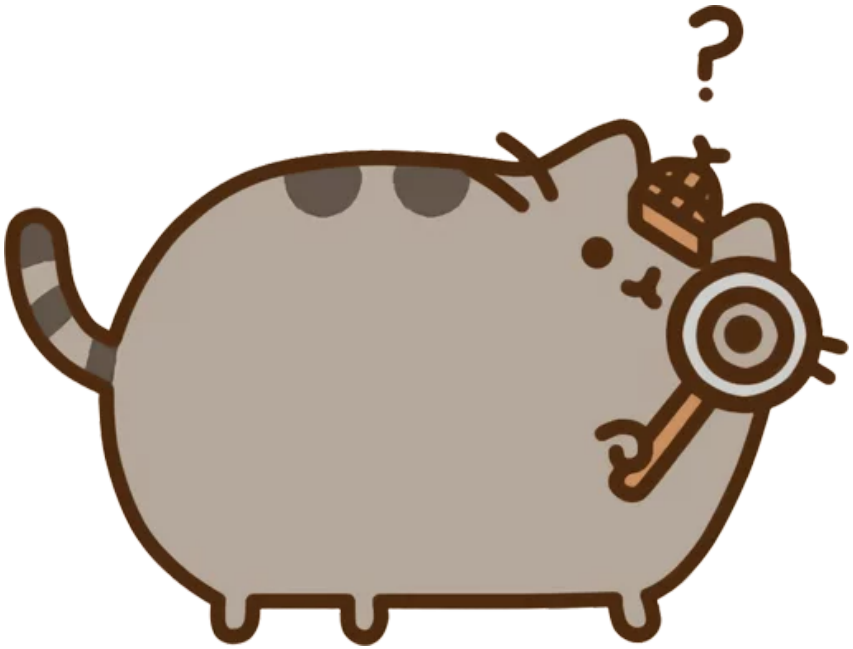
Last but not least...

https://turbo.fish
Bastion of The Turbofish https://github.com/rust-lang/rust/blob/master/src/test/ui/bastion-of-the-turbofish.rs

If we got time...

- 为什么方法里允许 `self: Pin<Self>`
- 为什么现在 Rust 标准库里的 HashTable 实现，在查找-插入的时候需要线性扫描两次？
- 为什么不能直接实现 Eq，必须得写一个 PartialEq？
- `impl Trait` 出现返回值、参数和别名内的意义有啥不一样？

That's All!



Question time!