程序君的 Rust 培训课

- Rust 初体验
- 所有权，借用检查，以及生命周期
- 类型系统和泛型编程
- 并发 - 原语
- 并发 - async/await
- 回顾第一讲
- 网络协议
- 网络安全
- 宏编程

培训中用到的工具

- Rust 1.52 (Rustup 安装)
- Rust search extension
- VSCode + rust analyzer
- Excalidraw

Rust 初体验

Experiment
Build
Hypothesis
Baseline
Feedback loop
Pivot
Preserve
Learn
Measure
Metrics
Analyze

为什么要有 Rust?

Rust 解决什么问题？给我们带来什么好处？

我们看看编程语言的 trade off

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

C

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- **Transparent**
- Velocity

Erlang/Elixir

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- **Resilliency**
- Rigor
- **Safety**
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Python

- **Approachability**
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

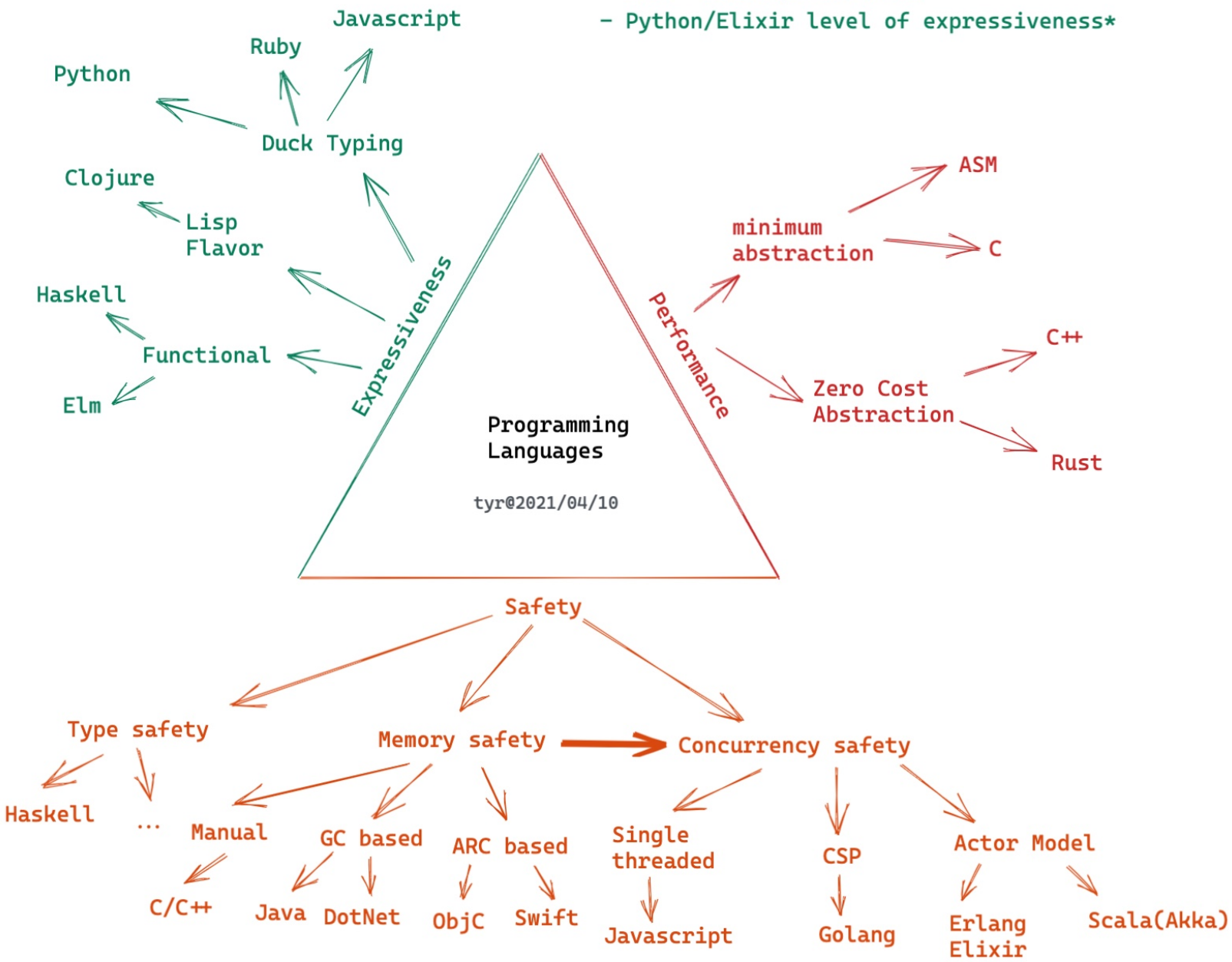
Java (in early days)

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- **Portability**
- Productivity
- Resilliency
- Rigor
- **Safety (memory)**
- **Security**
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

Rust

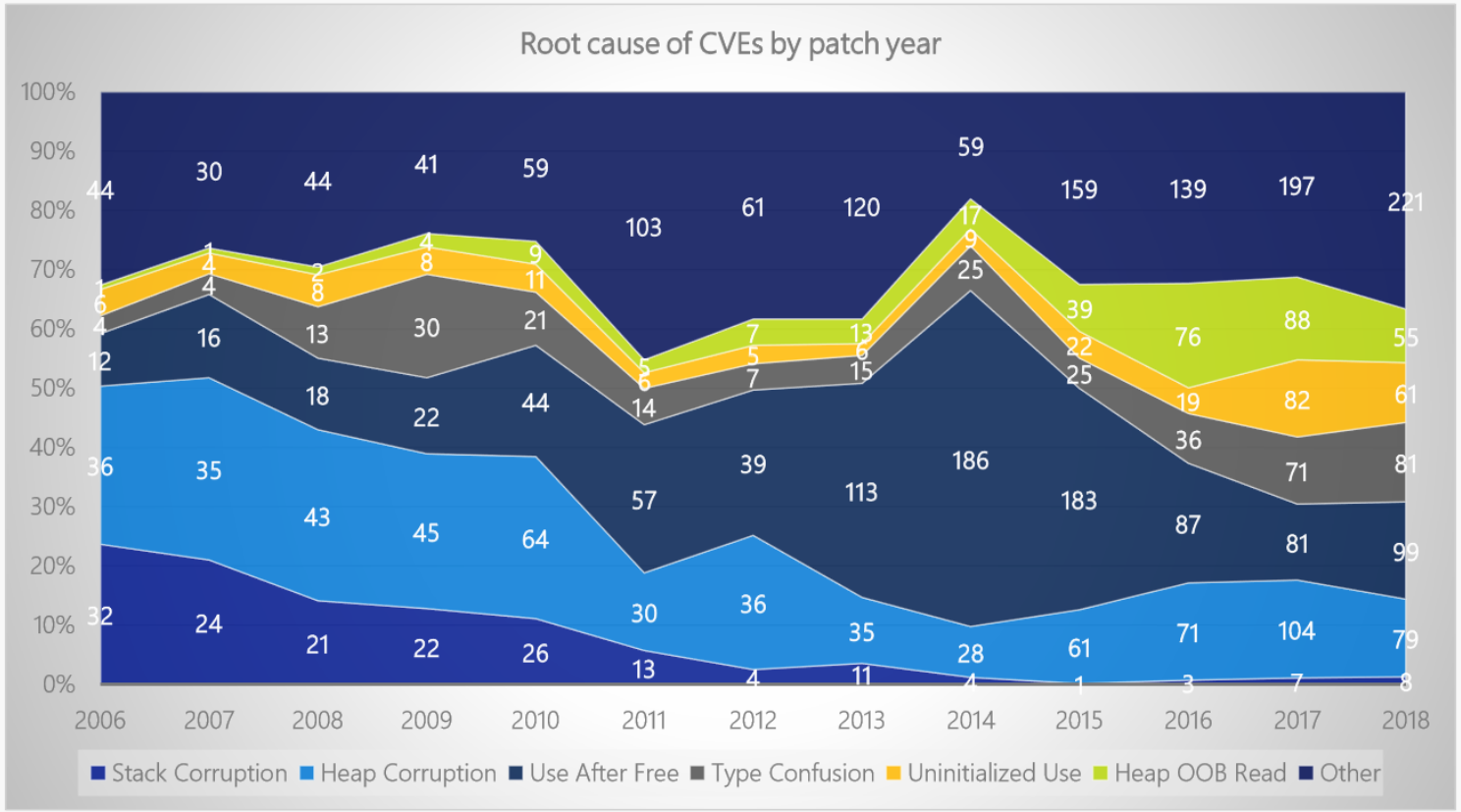
- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- Portability
- **Productivity**
- Resilliency
- Rigor
- **Safety!!!**
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

Rust:
- Type, Memory and Concurrency safety
- C/C++ level performance
- Python/Elixir level of expressiveness*



为什么语言安全性如此重要?

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

(来源: Memory Safety Issues Are Still the Leading Source of Security Vulnerabilities)

在编程语言级别达到安全性很困难！

- 内存安全并不容易（有很多 corner case）
- 并发安全非常困难（除非做很多妥协）
- 额外的抽象层级（意味着大量性能损失）

如何实现内存安全?

- 人工管理 - C/C++，很痛苦，容易出错
- 智能指针 - C++/ObjC/Swift：性能损失，循环引用问题
- GC - Java/DotNet/Erlang：大量的内存消耗，不必要的堆内存分配，以及（潜在的）STW
- Ownership - Rust：范式转换（paradigm shift），学习曲线

如何实现并发安全?

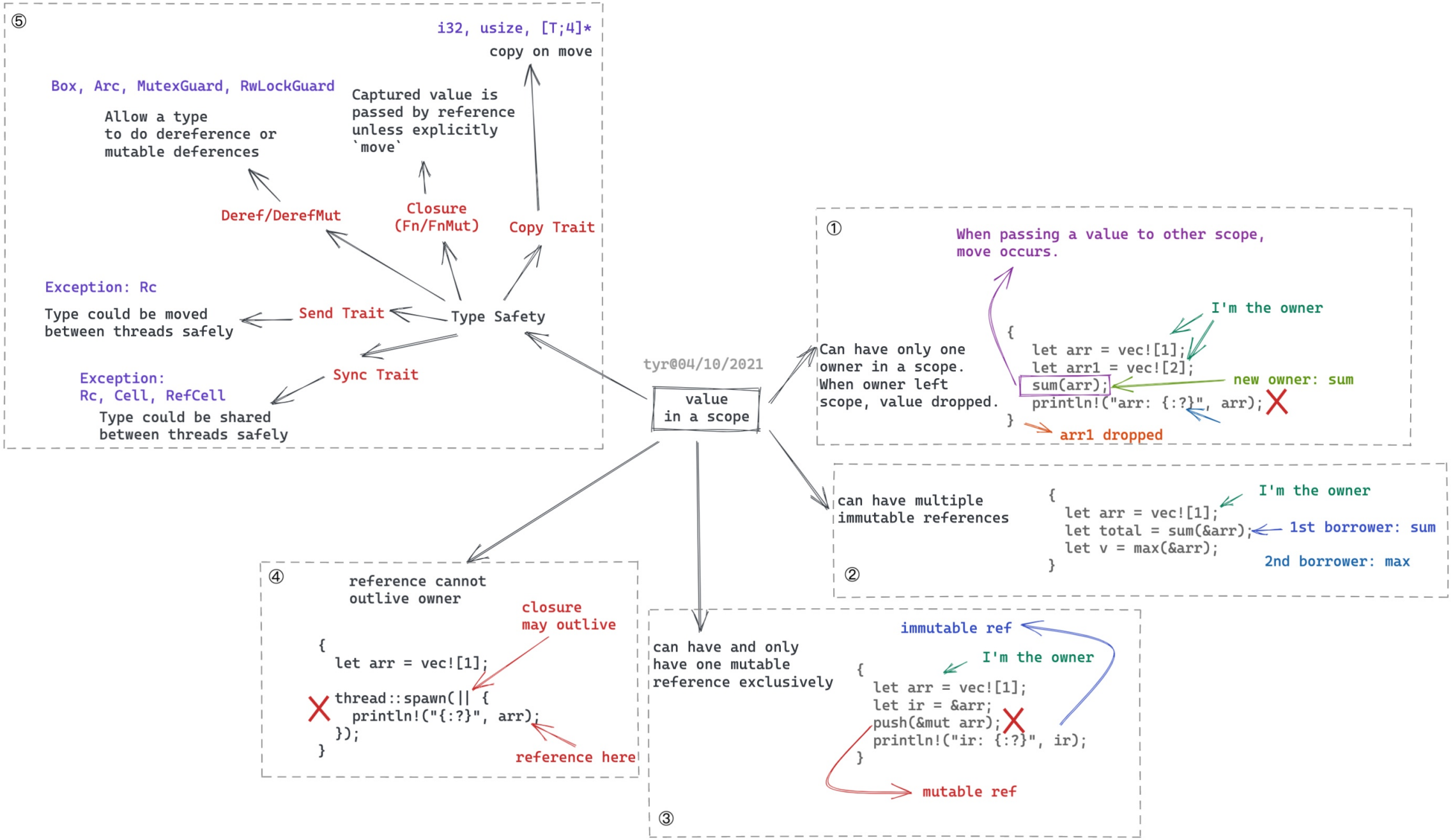
- 单线程 - javascript
  - 最安全的并发是单线程并发
  - 无法有效利用多核
- GIL - Python/Ruby
  - 一把大锁牺牲性能，换来安全
  - 锁粒度太大
- Actor model - Erlang/Akka
  - 通过消息同步（actor -> actor）
  - 额外内存拷贝和堆内存分配
- CSP - Golang
  - 通过消息同步（coroutine -> channel -> coroutine）
  - 额外内存拷贝和堆内存分配
- Ownership + Type System - Rust
  - 用类型安全来保证并发安全
  - 优雅，无性能损失，且和其它方案无缝兼容

Rust 如何实现 内存安全 和 并发安全?

并且不引入 额外开销?

我们从最基本的语法单元 — 值 说起

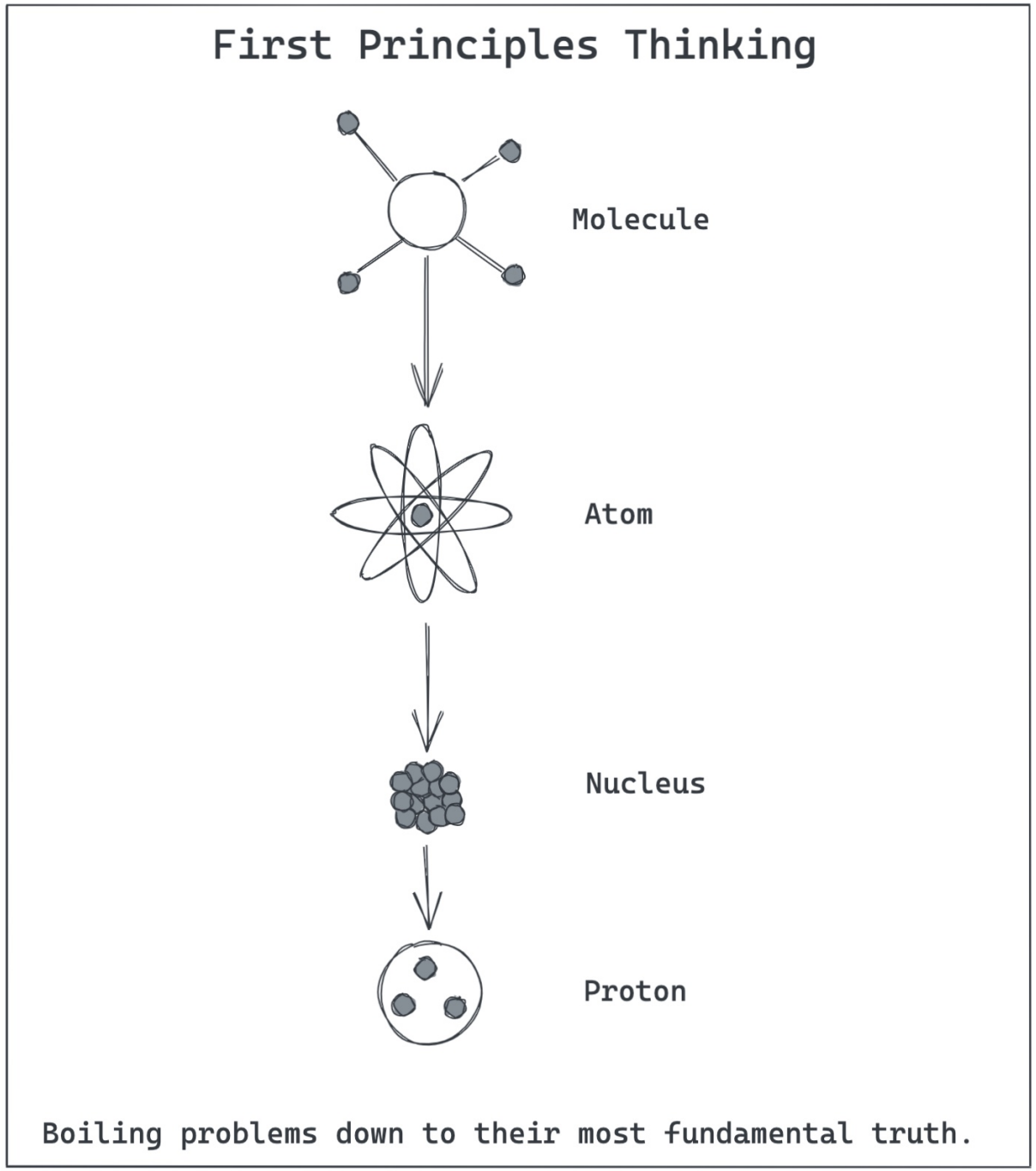
Live coding


⑤
i32, usize, [T;4]*
copy on move
Box, Arc, MutexGuard, RwLockGuard
Allow a type
to do dereference or
mutable deferences
Captured value is
passed by reference
unless explicitly
`move`
Deref/DerefMut
Closure
(Fn/FnMut)
Copy Trait
Exception: Rc
Type could be moved
between threads safely
Send Trait
Type Safety
Sync Trait
Exception:
Rc, Cell, RefCell
Type could be shared
between threads safely
tyr@04/10/2021
value
in a scope
①
When passing a value to other scope,
move occurs.
Can have only one
owner in a scope.
When owner left
scope, value dropped.
I'm the owner
{
let arr = vec![1];
let arr1 = vec![2];
sum(arr);
new owner: sum
println!("arr: {:?}", arr);
}
arr1 dropped
can have multiple
immutable references
{
I'm the owner
let arr = vec![1];
let total = sum(&arr);
1st borrower: sum
let v = max(&arr);
2nd borrower: max
}
②
④
reference cannot
outlive owner
closure
may outlive
{
let arr = vec![1];
thread::spawn(|| {
println!("{:?}", arr);
});
}
reference here
can have and only
have one mutable
reference exclusively
immutable ref
I'm the owner
{
let arr = vec![1];
let ir = &arr;
push(&mut arr);
println!("ir: {:?}", ir);
}
mutable ref
③

更多代码示例

```rust
fn main() {
    let mut arr: Vec<i32> = vec![1, 2, 3];    move occurs because `arr` has type `Vec<i32>`, which does not implement the `Copy` trait
    arr.push(4);

    let _result: Result<(), Error> = process(arr);    value moved here
    let _v: Option<i32> = arr.pop(); // failed since arr is moved    borrow of moved value: `arr`

    // you can have multiple immutable references
    let mut arr1: Vec<i32> = vec![1, 2, 3];
    let ir1: &Vec<i32> = &arr1;
    let ir2: &Vec<i32> = &arr1;    immutable borrow occurs here

    println!("ir1: {:?} ir2: {:?}", ir1, ir2);

    // but you can't have both mutable and immutable references
    let mr1: &mut Vec<i32> = &mut arr1;    cannot borrow `arr1` as mutable because it is also borrowed as immutable
    // let mr2 = &mut arr1;

    println!("mr1: {:?} mr2: {:?}", mr1, ir2);    immutable borrow later used here

    // by default, closure borrows the data
    let mut arr2: Vec<i32> = vec![1, 2, 3];
    thread::spawn(|| {    closure may outlive the current function, but it borrows `arr2`, which is owned by the current function
        arr2.push(4);    `arr2` is borrowed here
    });

    // we shall move the data explicitly
    let mut arr3: Vec<i32> = vec![1, 2, 3];
    thread::spawn(move || arr3.push(4));
}

fn thread_safety() {
    // but certain types cannot be moved to other thread safely
    let mut rc1: Rc<Vec<i32>> = Rc::new(vec![1, 2, 3]);
    thread::spawn(move || {    `Rc<Vec<i32>>` cannot be sent between threads safely
        rc1.push(4);
    });
}
```

```rust
fn thread_safety_reasoning() {
    let mut map: HashMap<&str, &str> = HashMap::new();    move occurs because `map` has type `HashMap<&str, &str>`, which does not implem
    map.insert(k: "hello", v: "world");

    // Arc is an atomic reference counter which can be moved safely across threads
    let mut ir: Arc<HashMap<&str, &str>> = Arc::new(data: map);    variable does not need to be mutable
    map.insert(k: "hello1", v: "world1"); // you can't do this since map is moved    borrow of moved value: `map`
    let ir1: Arc<HashMap<&str, &str>> = ir.clone(); // this is cheap, just reference counter clone
    thread::spawn(move || assert_eq!(ir1.get("hello"), Some(&"world")));
    // but arc is immutable, so this would fail
    thread::spawn(move || ir.insert(k: "hello2", v: "world2"));    cannot borrow data in an `Arc` as mutable

    // the compiler guide you to use types that provides mutable reference for threads

    // use Mutex - you can't clone a Mutex, thus you can't make it availabe for multiple threads
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Mutex<HashMap<&str, &str>> = Mutex::new(map1);
    let mr1 = mr.clone();    no method named `clone` found for struct `Mutex<HashMap<&str, &str>>` in the current scope
    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    mr1.lock().unwrap().insert("hello2", "world2");

    // use Mutex with Arc - now you have mutable access and multi-thread cloning
    let mut map2: HashMap<&str, &str> = HashMap::new();
    map2.insert(k: "hello", v: "world");
    let mr: Arc<Mutex<HashMap<&str, &str>>> = Arc::new(data: Mutex::new(map2));
    let mr1: Arc<Mutex<HashMap<&str, &str>>> = mr.clone();

    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    thread::spawn(move || mr1.lock().unwrap().insert(k: "hello2", v: "world2"));

    // can I use Box (smart pointer for heap allocation)?
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Arc<Box<HashMap<&str, &str>>> = Arc::new(data: Box::new(map1));
    let mr1: Arc<Box<HashMap<&str, &str>>> = mr.clone();
    thread::spawn(move || (**mr).insert(k: "hello1", v: "world1"));    cannot borrow data in an `Arc` as mutable
    mr1.insert(k: "hello2", v: "world2");    cannot borrow data in an `Arc` as mutable
}
```

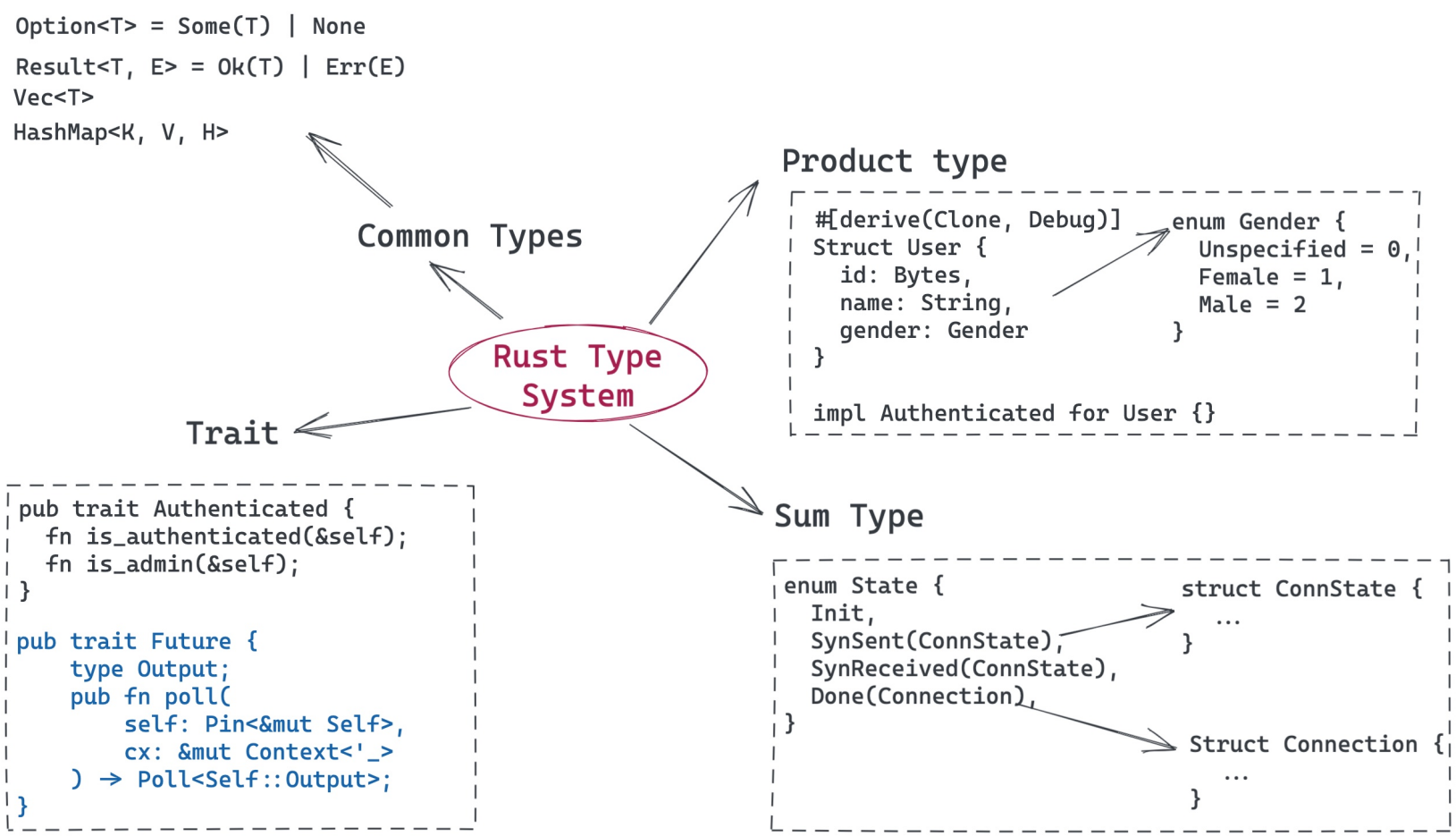


回顾

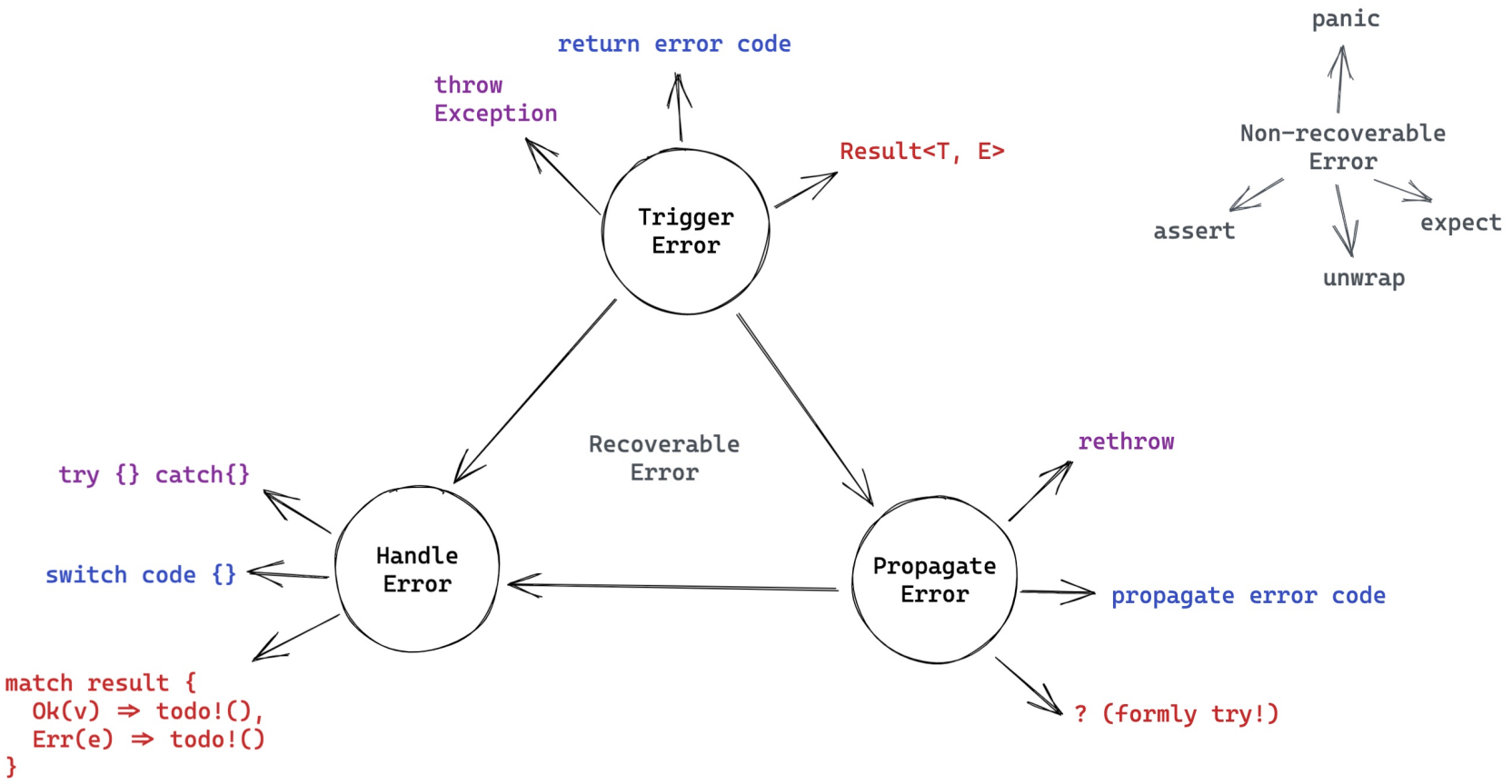
- 在一个 scope 中
  - 一个值只有一个所有者
  - 但可以有多个不可变引用
  - 以及唯一的可变引用（mutual exclusive）
  - 引用的生命周期不能超过值的生命周期
- 在多线程环境下
  - 类型安全（`Send` / `Sync`）保证并发安全

使用这些简单的规则，Rust 实现了 **零成本抽象** 的安全

类型系统


Option<T> = Some(T) | None
Result<T, E> = Ok(T) | Err(E)
Vec<T>
HashMap<K, V, H>
Common Types
Rust Type System
Trait
pub trait Authenticated {
fn is_authenticated(&self);
fn is_admin(&self);
}
pub trait Future {
type Output;
pub fn poll(
self: Pin<&mut Self>,
cx: &mut Context<'_>
) -> Poll<Self::Output>;
}
Product type
#[derive(Clone, Debug)]
Struct User {
id: Bytes,
name: String,
gender: Gender
}
impl Authenticated for User {}
enum Gender {
Unspecified = 0,
Female = 1,
Male = 2
}
Sum Type
enum State {
Init,
SynSent(ConnState),
SynReceived(ConnState),
Done(Connection),
}
struct ConnState {
...
}
Struct Connection {
...
}

错误处理


return error code
throw
Exception
Result<T, E>
Trigger
Error
panic
Non-recoverable
Error
assert
expect
unwrap
Recoverable
Error
rethrow
try {} catch{}
Handle
Error
switch code {}
Propagate
Error
propagate error code
match result {
Ok(v) ⇒ todo!(),
Err(e) ⇒ todo!()
}
? (formly try!)

Live coding: 用文件持久化数据结构

思考：如何在内存和 IO 设备间交换数据?

Rust 开发的效率如何?

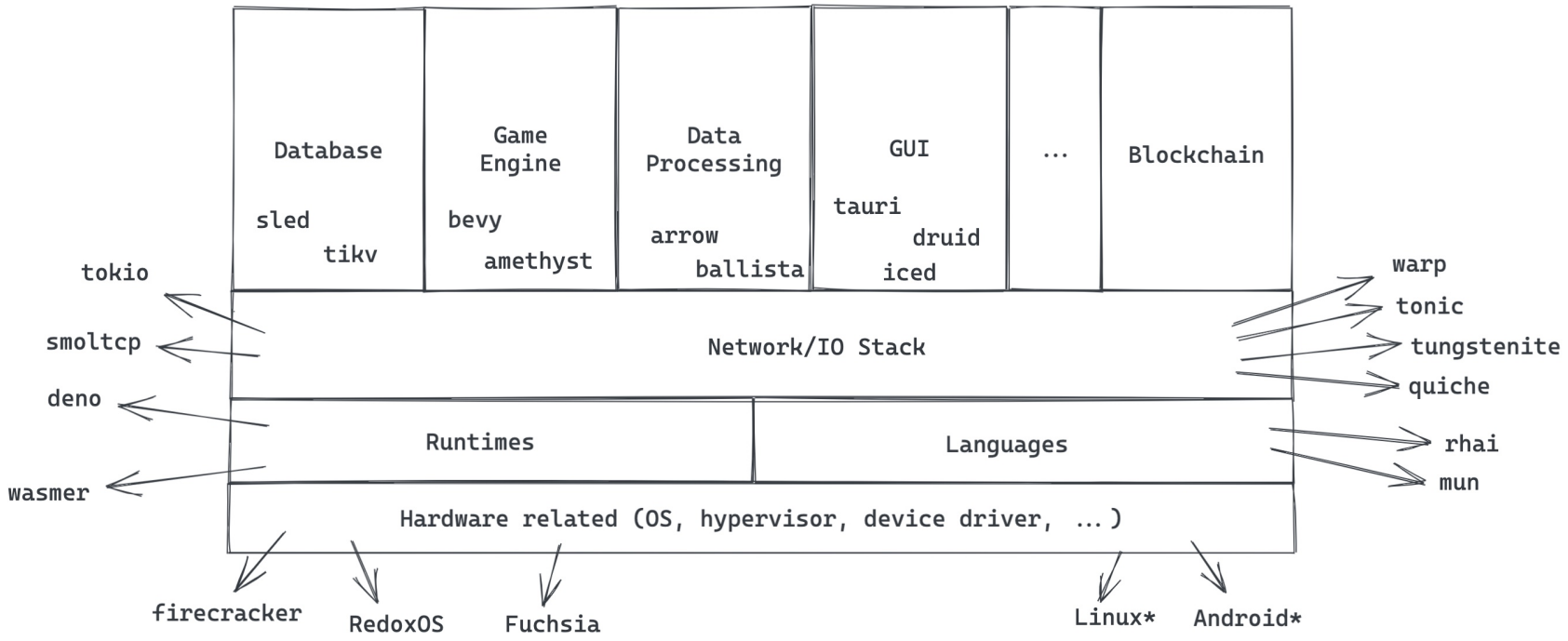
```toml
# client configuration

domain = "localhost"

[cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBeTCCASugAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMzEwMzEyMTg0NTU2WjA3MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xEjAQBgNVBAMMCURvbWFpbiBDQTAqMAUGAytlcAMhAAzhorM9IPsXjBTx
ZxykGl5xZrsj3X2XqKjaAVutnf7po1wwWjAUBgNVHREEDTALgglsb2NhbGhvc3Qw
HQYDVR0OBBYEFD+NqChBZDOs5FMgefHJSIWiRTHXMBIGA1UdEwEB/wQIMAYBAf8C
ARAwDwYDVR0PAQH/BAUDAwcGADAFBgMrZXADQQA9sligQcYGaBqTxR1+JadSelMK
Wp35+yhVVuu4PTL18kWdU819w3cVlRe/GHt+jjlbk1i22TvfO5AaNmdxySkO
-----END CERTIFICATE-----"""

# server configuration

[identity]
key = """-----BEGIN PRIVATE KEY-----
MFMCAQEwBQYDK2VwBCIEII0kozd0PJsbNfNUS/oqI/Q/enDiLwmdw+JUnTLpR9xs
oSMDIQAtkhJiFdF9SYBIMcLikWPRIgca/Rz9ngIgd6HuG6HI3g==
-----END PRIVATE KEY-----"""

[identity.cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBazCCAR2gAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMjIwMzE0MTg0NTU2WjA5MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xFDASBgNVBAMMC0dSUEMgU2VydmVyMCowBQYDK2VwAyEALZISYhXRfUmA
SDHC4pFj0SIHGv0c/Z4CIHeh7huhyN6jTDBKMBQGA1UdEQQNMAuCCWxvY2FsaG9z
dDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMEBTADAQEAMA8GA1UdDwEB/wQF
AwMH4AAwBQYDK2VwA0EAy7EOIZp73XtcqaSopqDGWU7Umi4DVvIgjmY6qbJZP0sj
ExGdaVq/7MOlZl1I+vY7G0NSZWZIUilX0CoOkrn0DA==
-----END CERTIFICATE-----"""
```

```rust
// you could also build your config with cert and identity separately. See tests.
let config: ServerTlsConfig = toml::from_str(config_file).unwrap();
let acceptor = config.tls_acceptor().unwrap();
let listener = TcpListener::bind(addr).await.unwrap();
tokio::spawn(async move {
    loop {
        let (stream, peer_addr) = listener.accept().await.unwrap();
        let stream = acceptor.accept(stream).await.unwrap();
        info!("server: Accepted client conn with TLS");

        let fut = async move {
            let (mut reader, mut writer) = split(stream);
            let n = copy(&mut reader, &mut writer).await?;
            writer.flush().await?;
            debug!("Echo: {} - {}", peer_addr, n);
        }

        tokio::spawn(async move {
            if let Err(err) = fut.await {
                error!("{:?}", err);
            }
        });
    }
});
```

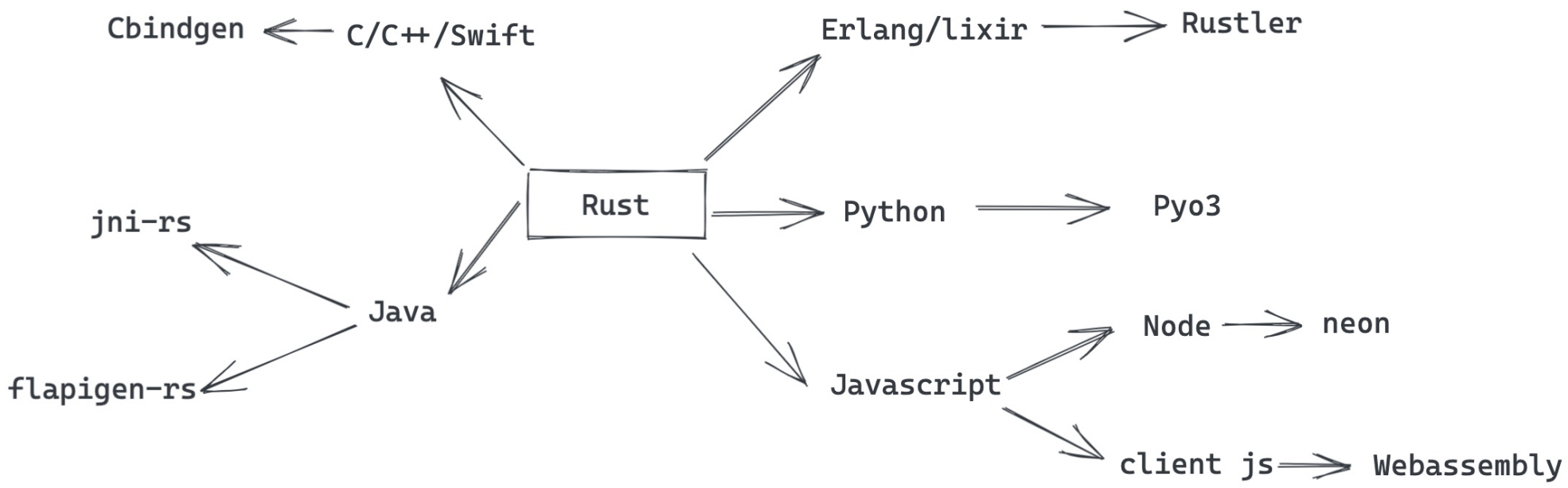
Client: You, a month ago • init the project

```rust
let msg = b"Hello world\n";
let mut buf = [0; 12];

// you could also build your config with cert and identity separately. See tests.
let config: ClientTlsConfig = toml::from_str(config_file).unwrap();
let connector = config.tls_connector(Uri::from_static("localhost")).unwrap();

let stream = TcpStream::connect(addr).await.unwrap();
let mut stream = connector.connect(stream).await.unwrap();
info!("client: TLS conn established");

stream.write_all(msg).await.unwrap();

info!("client: send data");

let (mut reader, _writer) = split(stream);

reader.read_exact(buf).await.unwrap();

info!("client: read echoed data");
```

Rust 可以用来做什么?



Rust 和其它语言的互操作

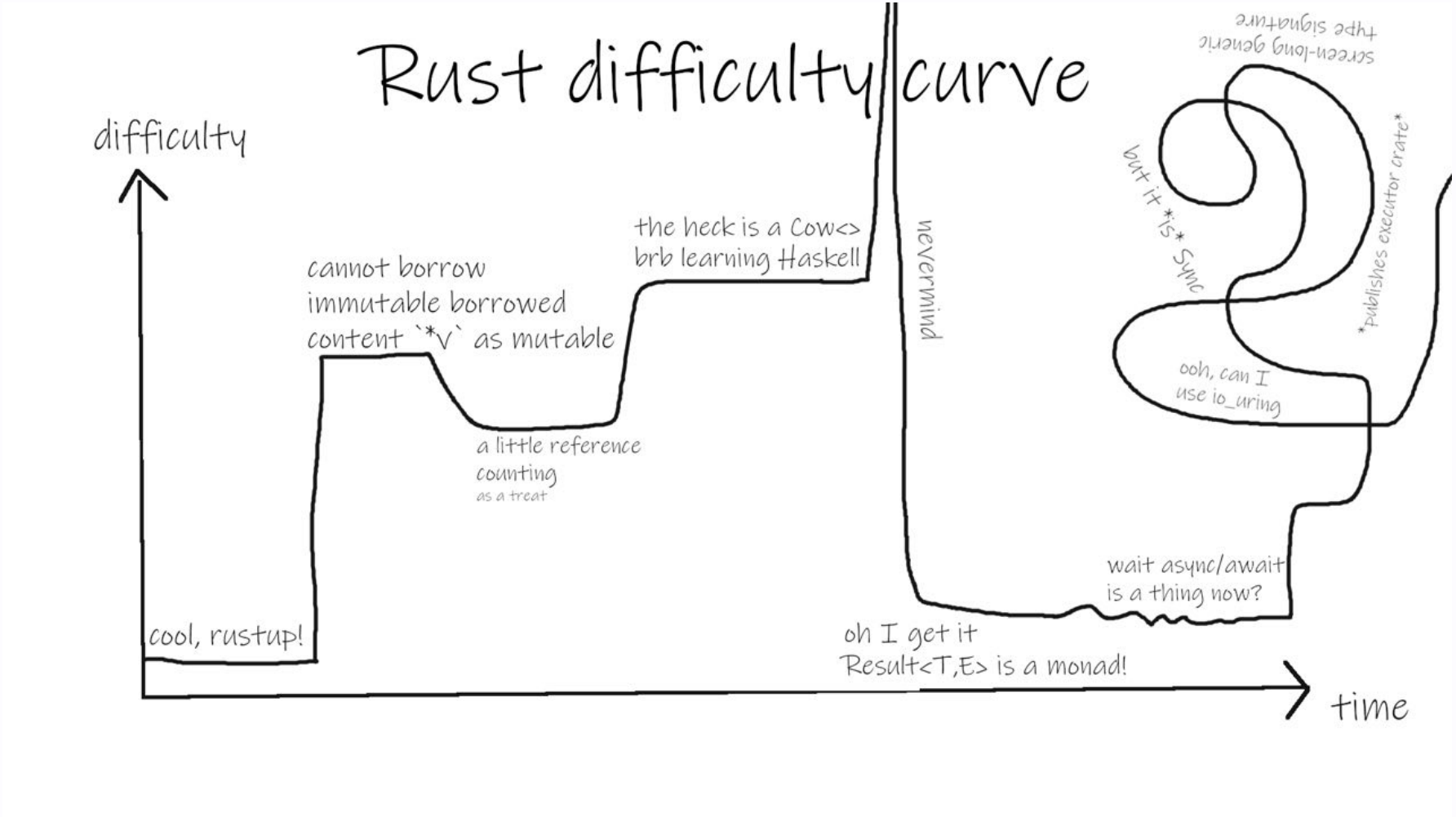


我是个 xyz 程序员，学 Rust 该注意什么？

- Erlang/Elixir: 所有权，类型系统，可变性
- Java/Swift/Scala: 所有权
- Typescript: 所有权，多线程编程
- Python: 所有权，类型系统，并发处理

学习 Rust 的误区...

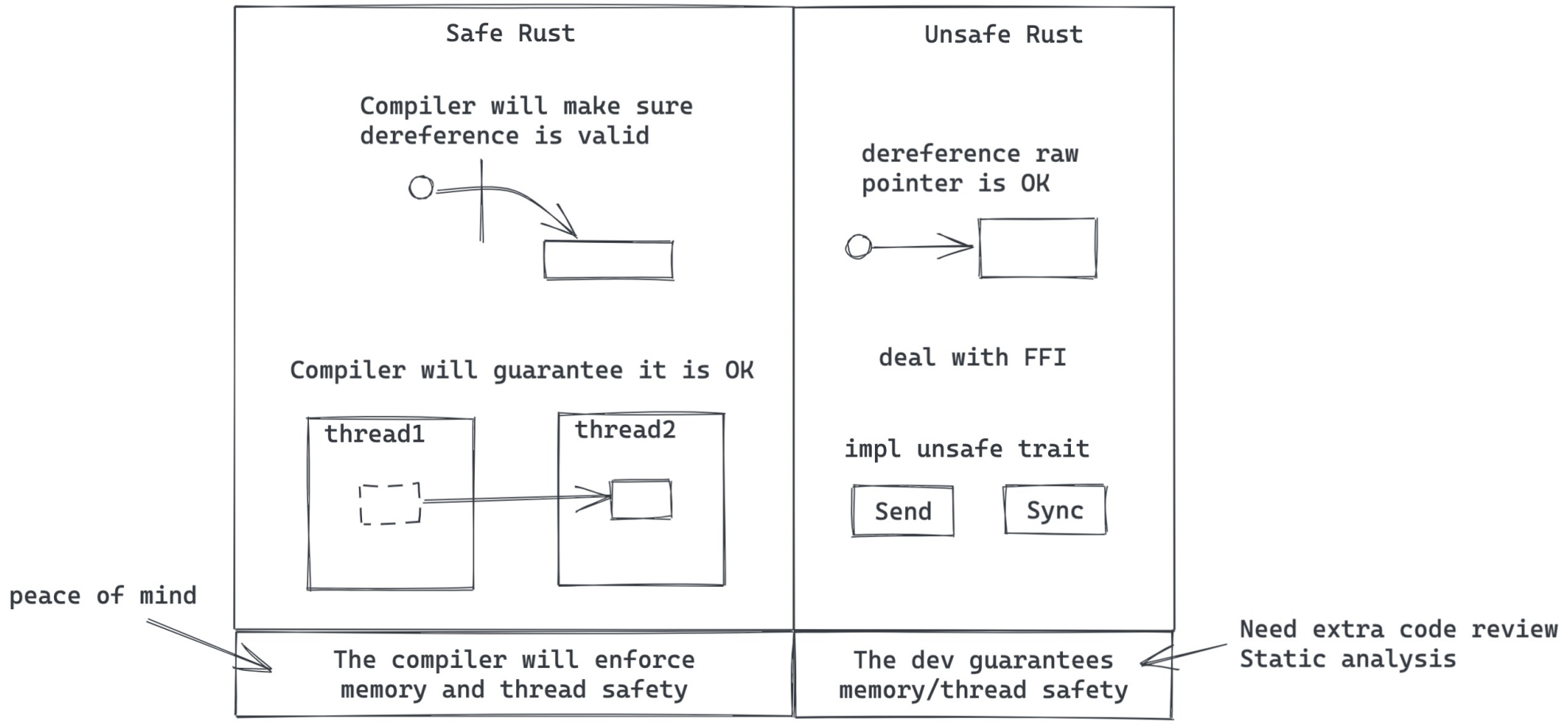
1. Rust 非常难学?


Rust difficulty curve
difficulty
time
cool, rustup!
cannot borrow immutable borrowed content `*v` as mutable
a little reference counting
as a treat
the heck is a Cow<>
brb learning Haskell
nevermind
oh I get it
Result<T,E> is a monad!
wait async/await is a thing now?
ooh, can I use io_uring
but it *is* Sync
publishes executor crate
screen-long generic type signature

非也，Rust 就是想跟你处得明明白白

- 范式转移（Paradigm Shift）
- 大量计算机体系结构操作系统相关的知识一下子扑鼻而来
- 如果学习一门语言的痛苦是 100%：
  - Rust：
    - 编译器帮你降低了 10% 的痛苦
    - 然而你需要在头 3-6 个月承受剩下 70% 的痛苦
    - 在接下来 3-5 年承受最后 20% 的痛苦
  - 其它语言：
    - 头 3-6 个月，你只需要承受 10-30% 的痛苦
    - 之后的 3-5 年你可以慢慢承受 70-90% 的痛苦

2. Unsafe Rust 听上去好可怕！

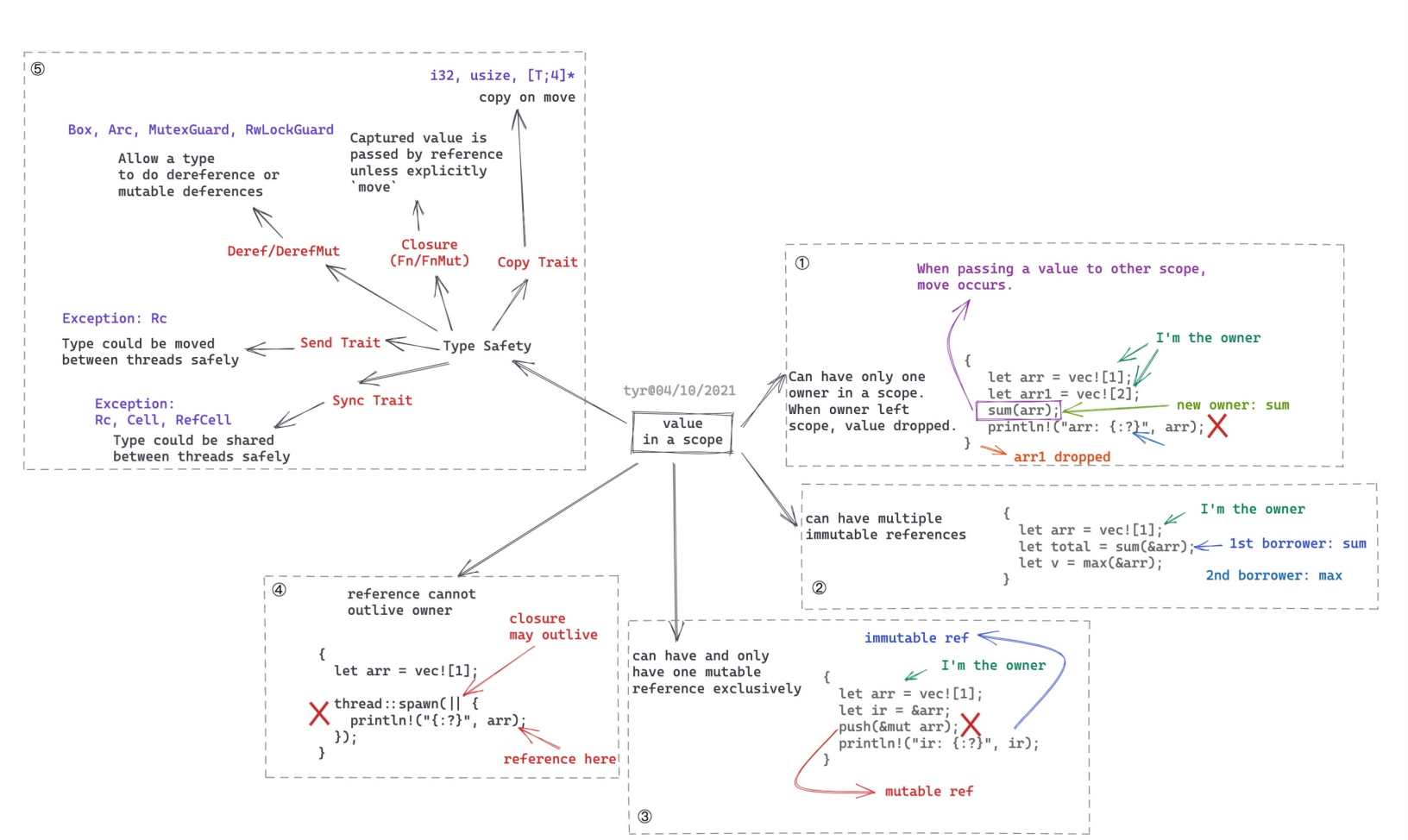


参考资料

- The pain of real linear types in Rust
- Substructural type system
- Rust official book
- Rust official site
- Awesome Rust
- Are we web yet?
- Are we async yet?
- Are we gui yet?
- Are we learning yet?
- Are we game yet?
- Are we quantum yet?
- Are we IDE yet?
- Rust is for Professionals

所有权，借用检查，以及生命周期

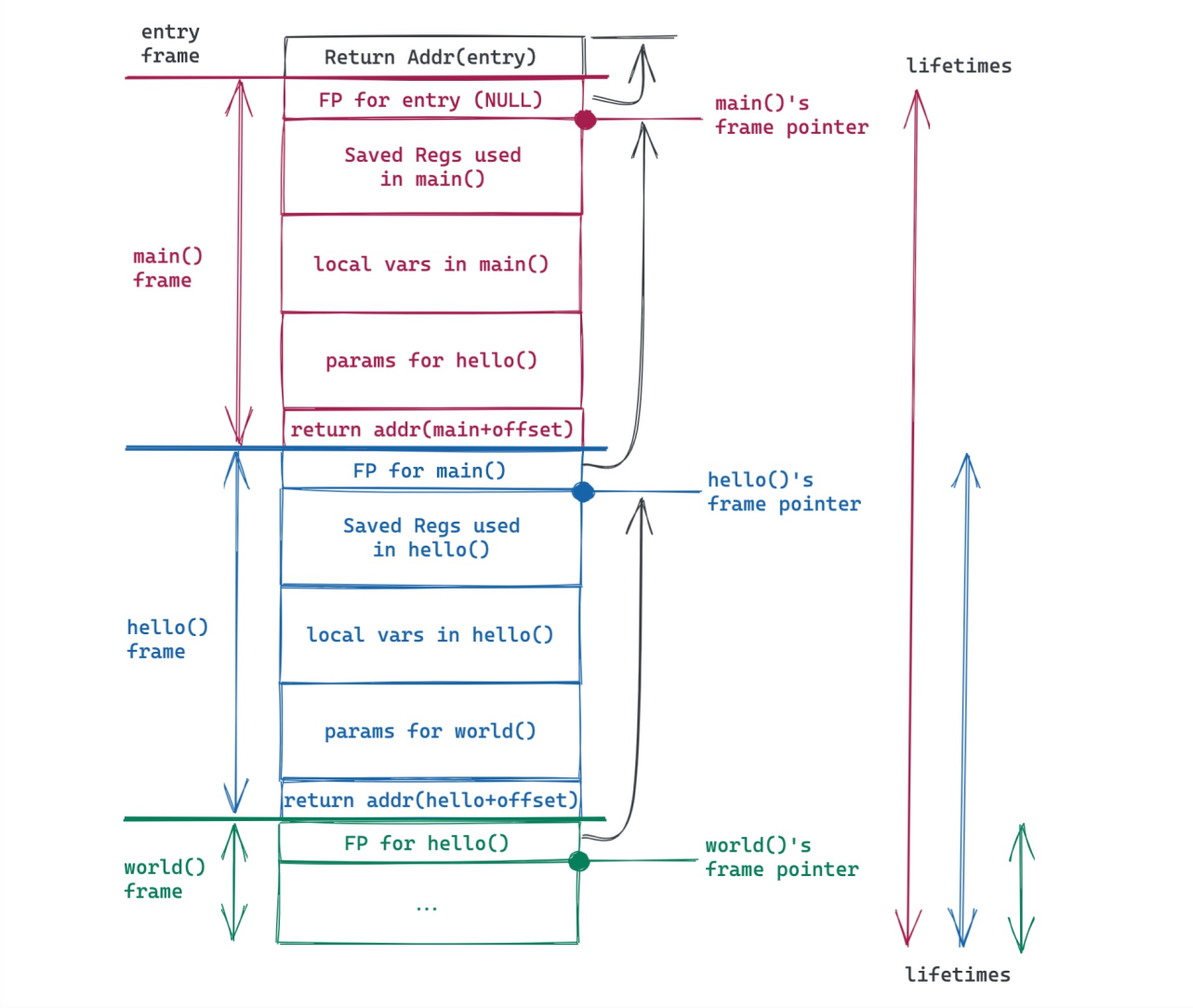
再探所有权和借用检查


⑤
i32, usize, [T;4]*
copy on move
Box, Arc, MutexGuard, RwLockGuard
Allow a type
to do dereference or
mutable deferences
Captured value is
passed by reference
unless explicitly
`move`
Deref/DerefMut
Closure
(Fn/FnMut)
Copy Trait
Exception: Rc
Type could be moved
between threads safely
Send Trait
Type Safety
Sync Trait
Exception:
Rc, Cell, RefCell
Type could be shared
between threads safely
tyr@04/10/2021
value
in a scope
①
When passing a value to other scope,
move occurs.
Can have only one
owner in a scope.
When owner left
scope, value dropped.
I'm the owner
{
let arr = vec![1];
let arr1 = vec![2];
sum(arr);
new owner: sum
println!("arr: {:?}", arr);
}
arr1 dropped
can have multiple
immutable references
{
let arr = vec![1];
let total = sum(&arr);
let v = max(&arr);
}
I'm the owner
1st borrower: sum
2nd borrower: max
②
④
reference cannot
outlive owner
closure
may outlive
{
let arr = vec![1];
thread::spawn(|| {
println!("{:?}", arr);
});
}
reference here
can have and only
have one mutable
reference exclusively
immutable ref
I'm the owner
{
let arr = vec![1];
let ir = &arr;
push(&mut arr);
println!("ir: {:?}", ir);
}
mutable ref
③

生命周期（Lifetime）并非新概念

栈内存的生命周期管理

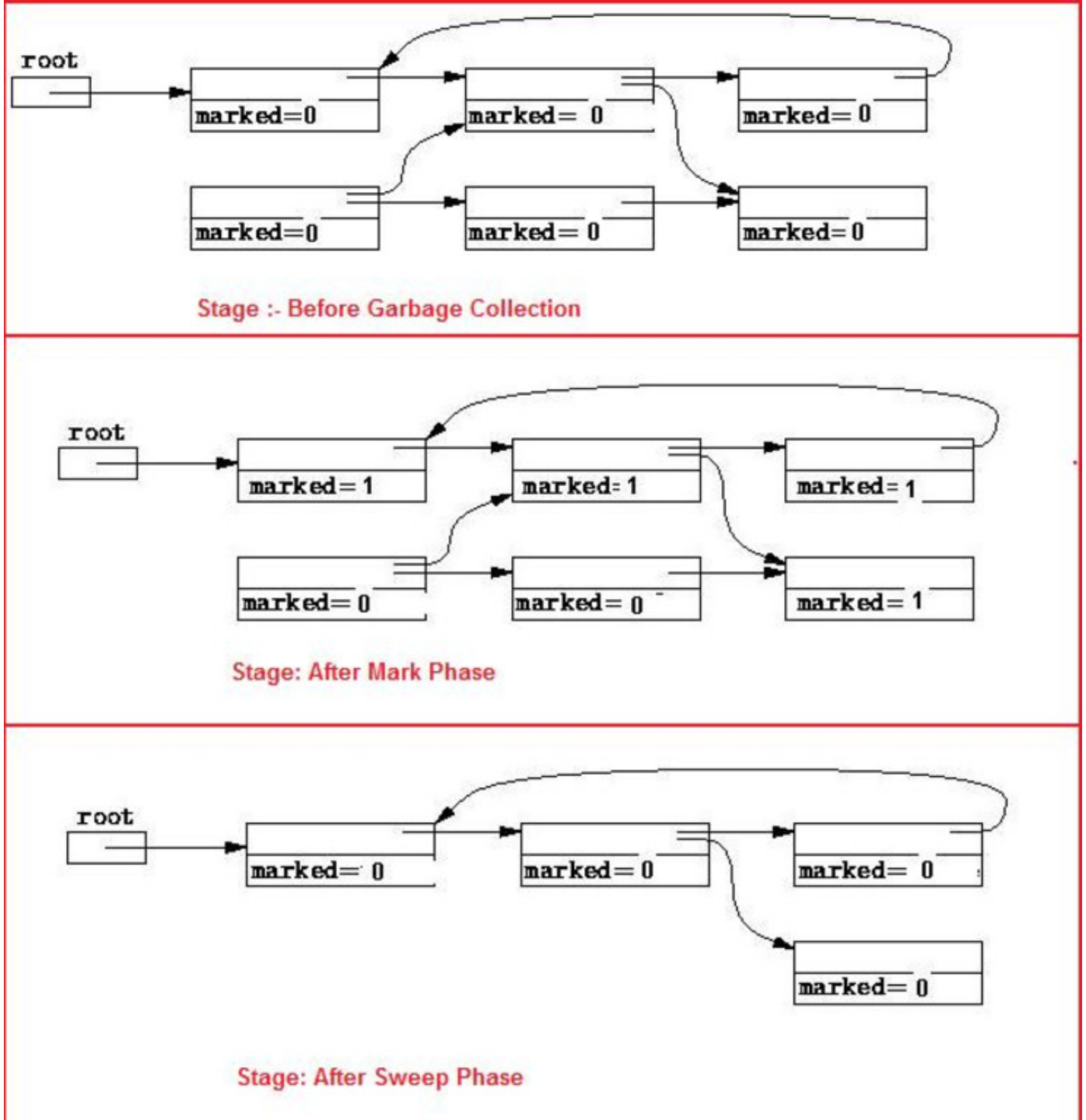
```c
#include <stdio.h>
static int VALUE = 42;

void world(char *st, int num) {
    printf("%s(%d)\n", st, num);
}


void hello(int num) {
    char *st = "hello world";
    int v = VALUE+num;
    world(st, v);
}

int main() {
    hello(2);
}
```



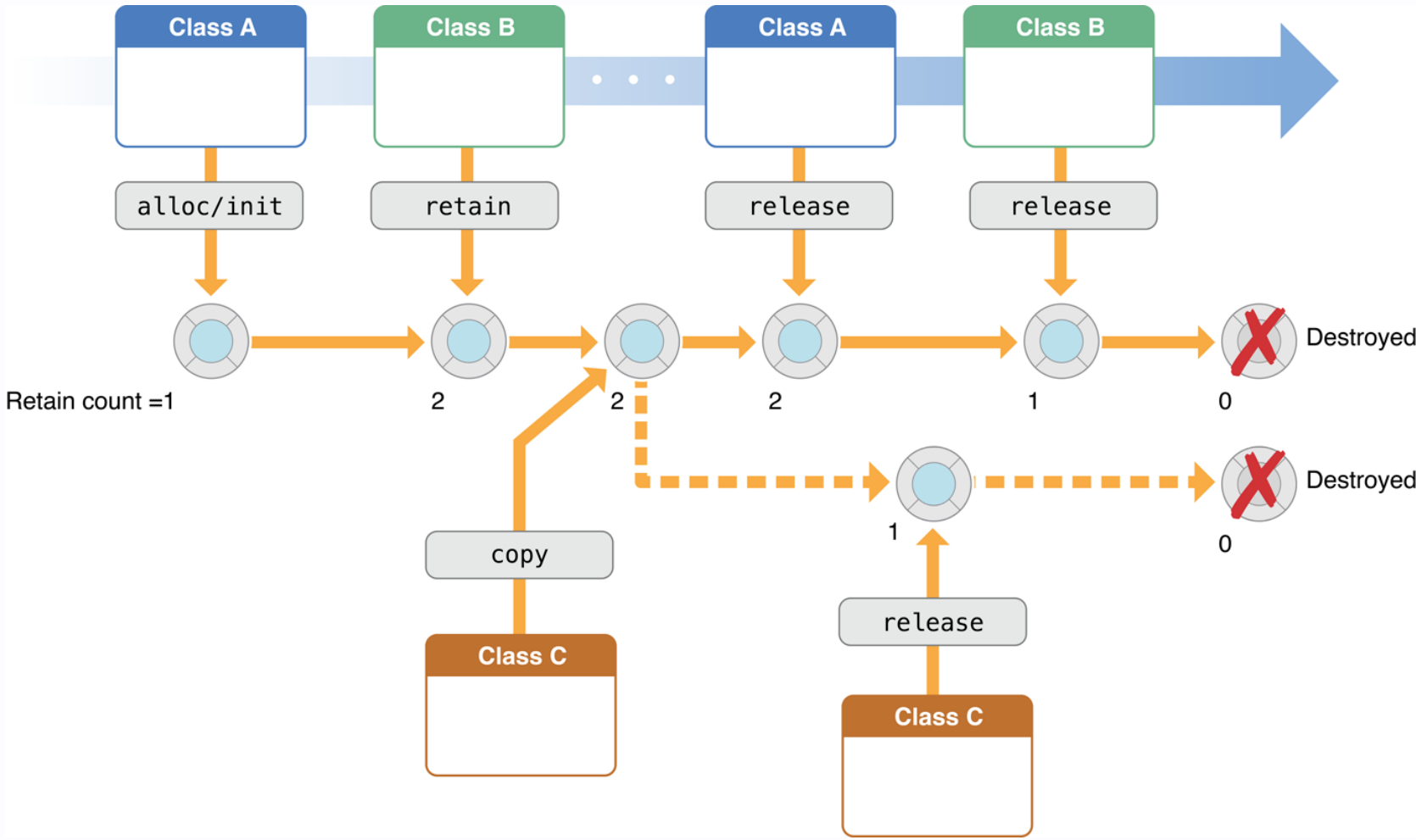
堆内存的生命周期管理

手工管理，GC，ARC，...

Tracing GC


root
marked=0
marked= 0
marked= 0
marked=0
marked= 0
marked= 0
Stage :- Before Garbage Collection
root
marked= 1
marked= 1
marked= 1
marked= 0
marked= 0
marked= 1
Stage: After Mark Phase
root
marked= 0
marked= 0
marked= 0
marked= 0
Stage: After Sweep Phase

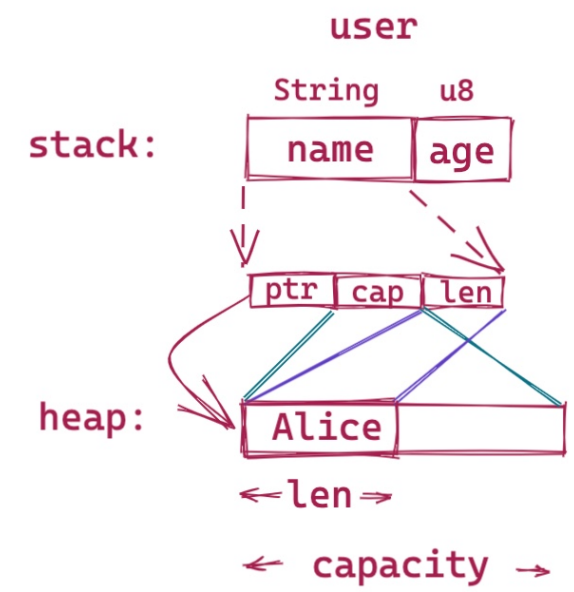
ARC


Class A
Class B
Class A
Class B
alloc/init
retain
release
release
Retain count =1
2
2
2
1
0
Destroyed
1
0
Destroyed
copy
release
Class C
Class C

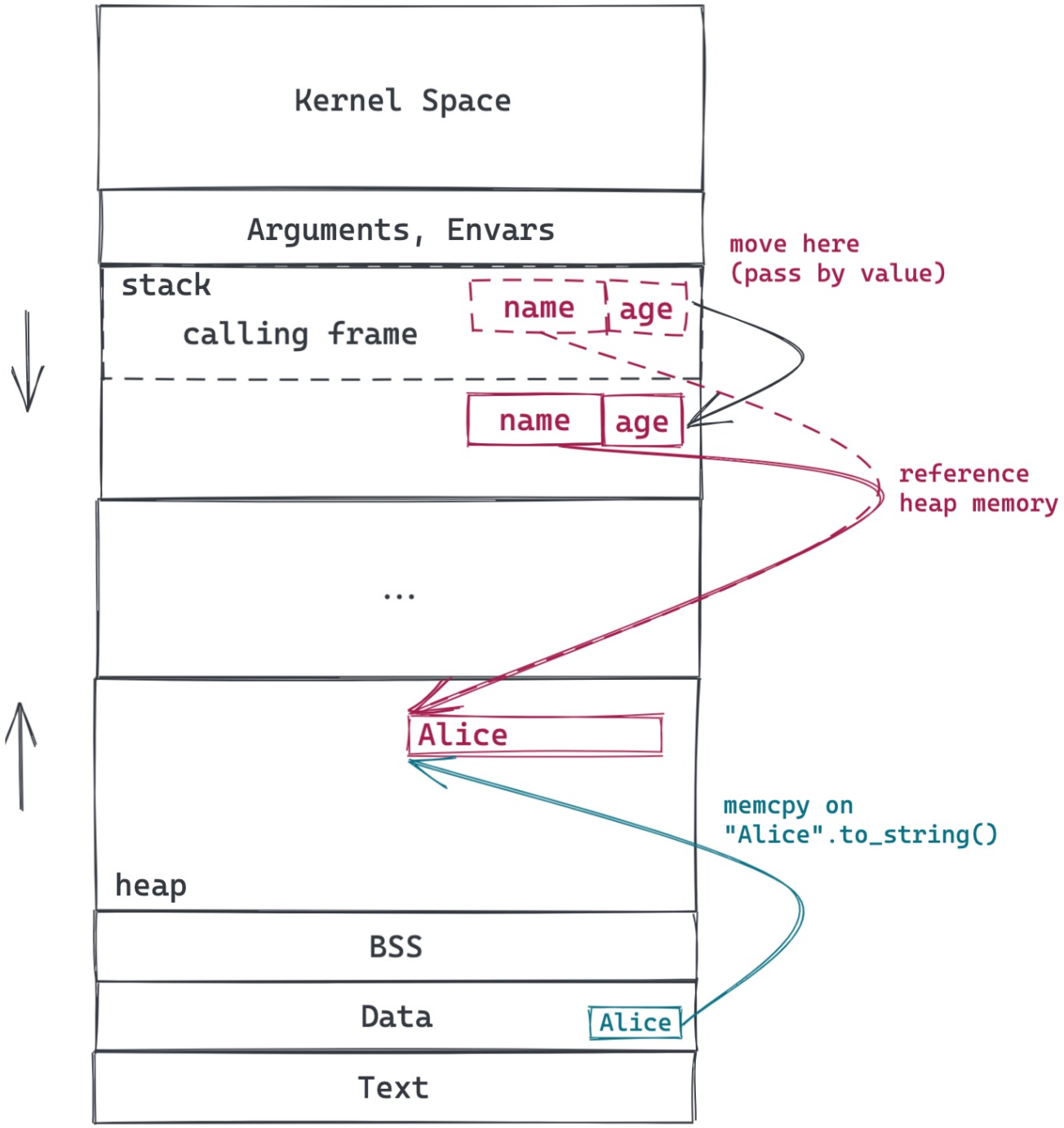
Rust 如何打破常规?

Rust 并非独创，它继承了 C++ 和 Cyclone 的很多研究

Move 语义：保证有且只有一个 owner

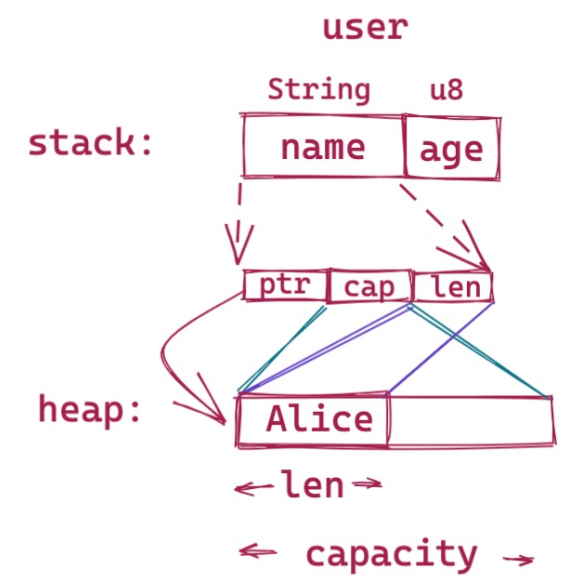



```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(user);
   ...
}
```

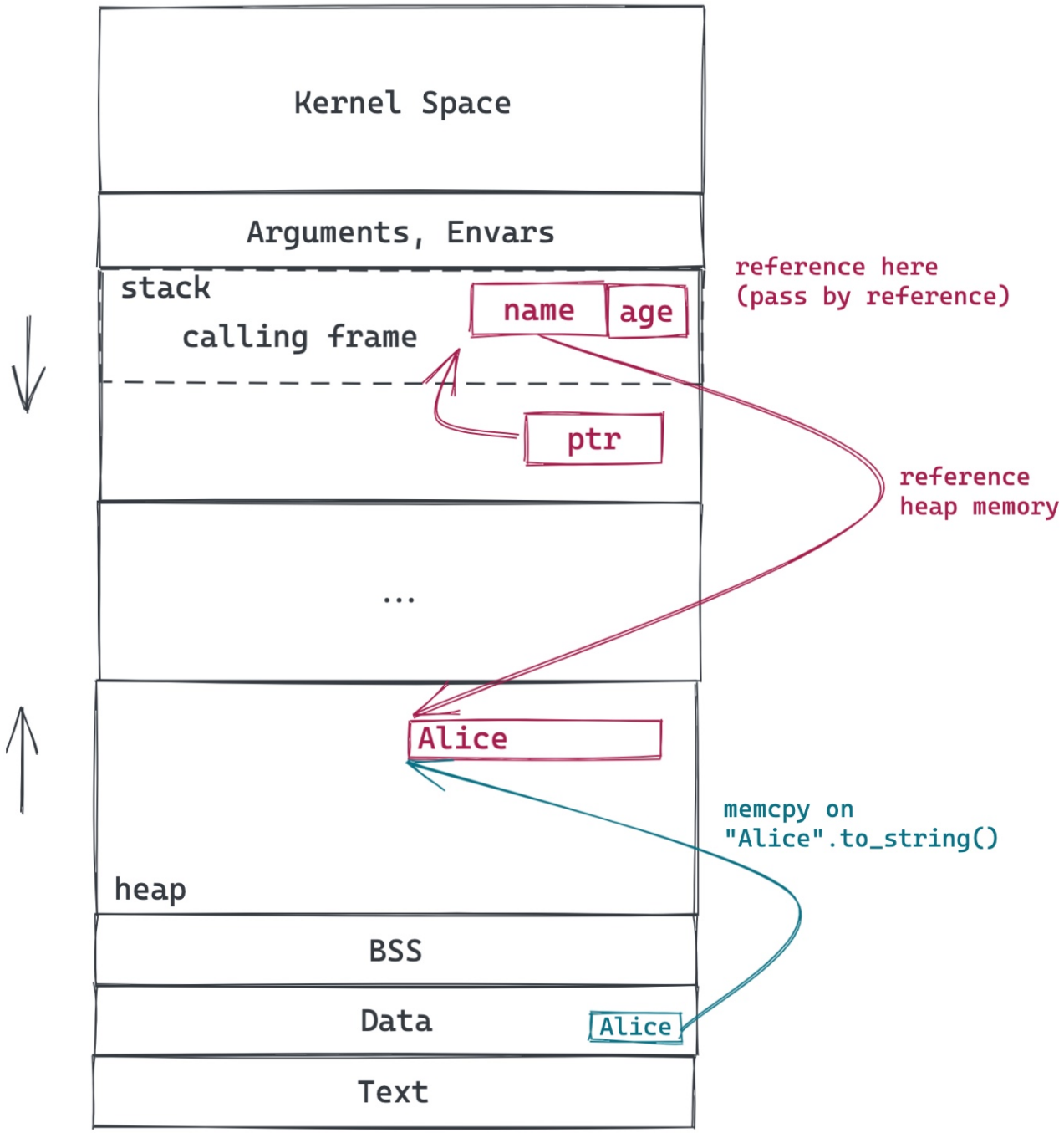


Borrow 语义：共享下的线程安全

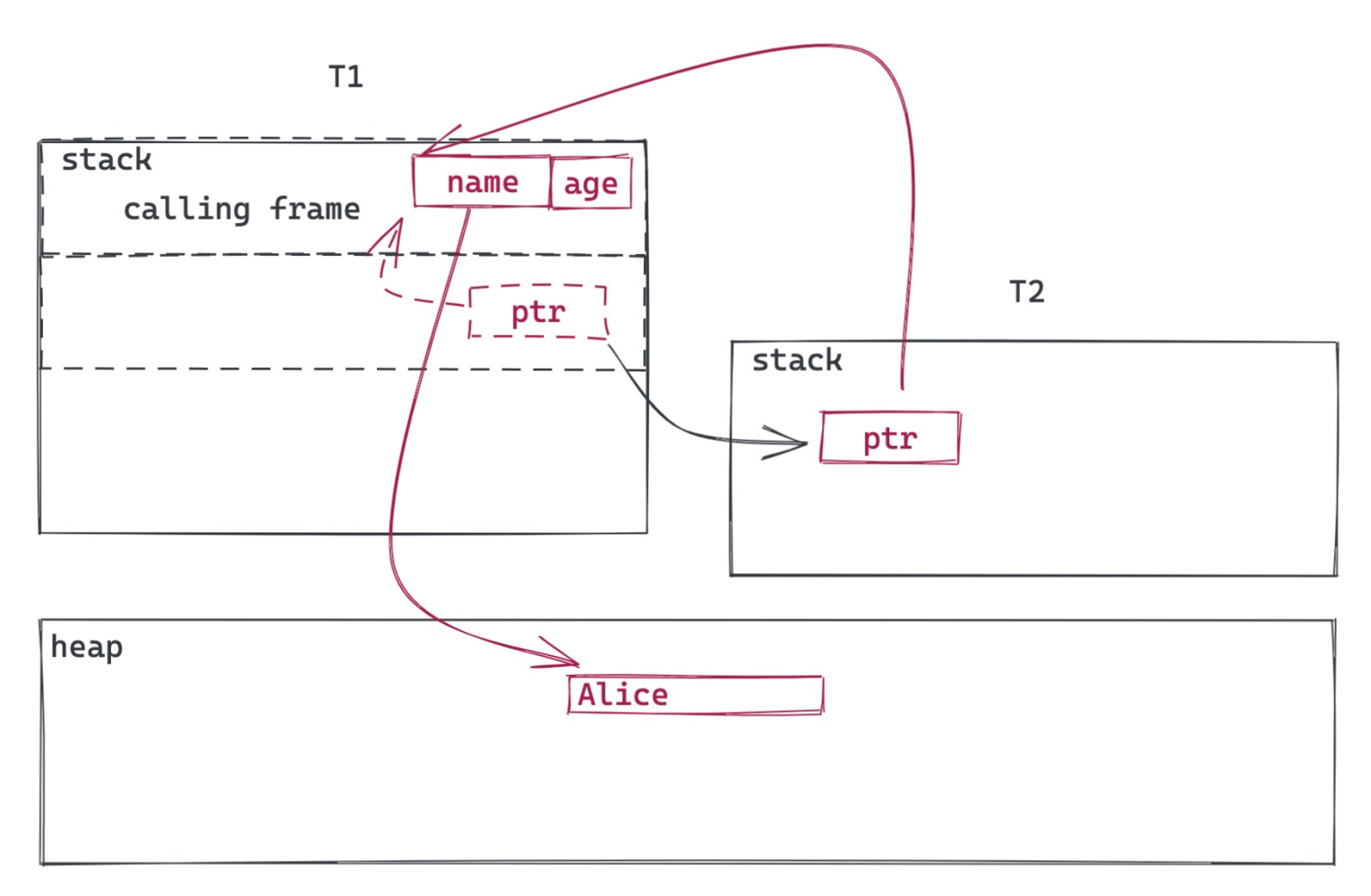
思考题：为什么同样使用了引用，在 Rust 可以保证线程安全，而 C++ 不行?




```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(&user);
   ...
}
```



Rust 生命周期检查器如何避免有问题的引用?



带生命周期限制的借用

- 可以借用任何值（栈内存，堆内存）
- 编译期检查（不需要耗费运行期的 CPU）
- Rust 借用检查器基本上是个生命周期检查器

我们如何告诉编译器生命周期?

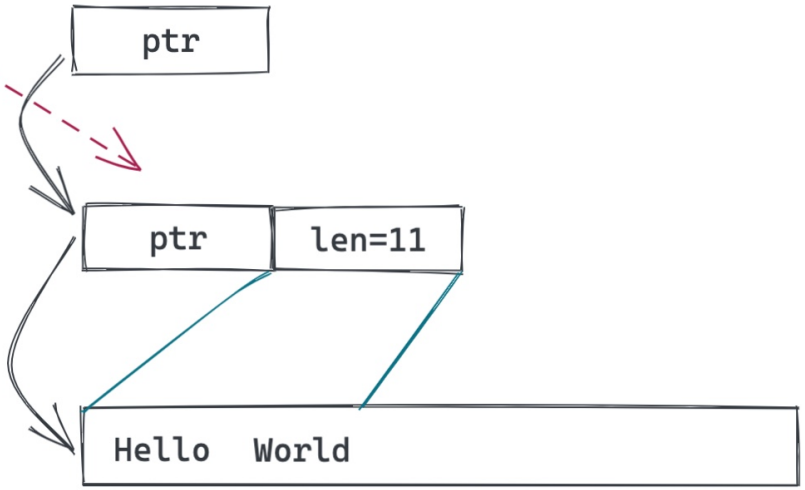
- 类似泛型，但有其专用符号 `'`
- 只有生命周期无法被推断出来，才需要声明

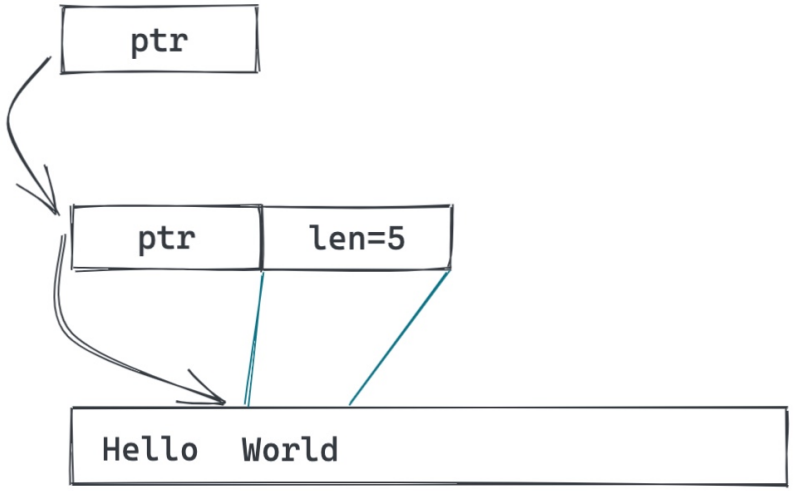
```
// need explicit lifetime
struct User<'a> {
    name: &'a str,
    ...
}
fn process<T, 'a, 'b>(item1: &'a T, item2: &'b T) {}

// &'a User could be written as &User since on confliction
fn lifetime_example(user: &User) { // --+ Lifetime 'a
    if user.is_authed() {          //   |--+ Lifetime 'b
        let role = user.roles();   //   |  |
                                   //   |  |--+ Lifetime 'c
        verify(&role);             //   |  |  |
                                   //   |  |--+
    }                              //   |--+
}                                  // --+

fn verify(x: &Role) { /*...*/ }
```

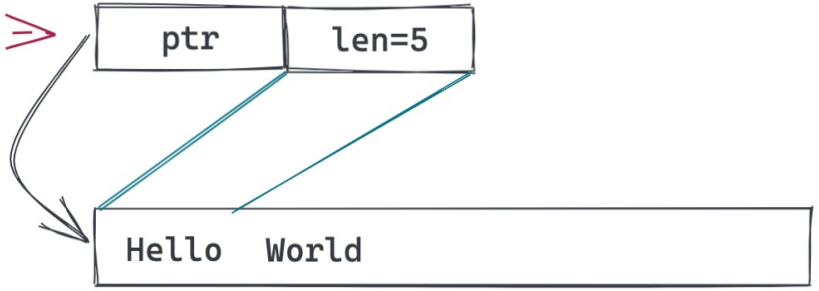
Live coding: strtok

one of 'a can be ommitted

```
pub fn strtok<'a>(s: &'a mut &'a str, delimiter: char)
  -> &'a str {
    if let Some(i) = s.find(delimiter) {
        let prefix = &s[..i];
        let suffix = &s[(i + delimiter.len_utf8())..];
        *s = suffix;
        prefix
    } else {
        let prefix = *s;
        *s = "";
        prefix
    }
}

fn it_works() {
    let x1 = "hello world".to_owned();
    let mut x = x1.as_str();
    let hello = strtok(&mut x, ' ');
    assert_eq!(hello, "hello");
    // assert_eq!(x, "world");
}
```







Static Lifetime

- `'static`
- 在 bss / data / text section 中的数据
  - constants / static variables
  - string literals
  - functions
- 如果在类型的 trait bound 中使用:
  - 类型不包含任何非静态的引用（non-static references）
- 有所有权的数据的 lifetime bound 是 `'static`，引用数据不是

我们再回头看：Thread spawn

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
```

这里 `'static` lifetime bound 是说： `F` 不能使用任何借用的数据

RAII (Resource Acquisition Is Initialization)

- 对象的初始化会保证资源的初始化
- 对象的释放会保证资源的释放

Drop Trait

- memory
- file
- socket
- lock
- any other OS resources

RAII live coding: 博物馆门票

疫情期间，博物馆内限流最大容量 50 人，满了之后，出来一个才能进一个，怎么设计?

总结：Cost of defects

- 代码中不引入任何缺陷（只要是人，就无法避免错误）
- 在敲入代码的时候就能被清晰地告知代码中的缺陷（发现问题时间：毫秒级）
- 在编译或者单元测试时发现（秒级/分钟级）
- 代码被 push 到 PR 中，CI 发现错误（分钟级/天级）
- 别人 review 你的代码时发现问题（小时级/天级）
- 代码 merge 到 master，更严格的 CI（比如 end-to-end test）发现错误（小时级/天级）
- 代码被部署后回归测试发现问题（天/周/月）
- 代码被部署后很久用户发现问题（周/月/年）

参考资料

- Mark-And-Sweep (Garbage Collection Algorithm)
- Tracing garbage collection
- Swift: Avoiding Memory Leaks by Examples
- Reference counting
- Fearless concurrency with Rust
- Rust means never having to close a socket
- Programming Rust: ownership
- Crust of Rust: lifetime annotation (recommended)

类型系统和泛型编程

Rust Type System

Common Types

```
Option<T> = Some(T) | None
Result<T, E> = Ok(T) | Err(E)
Vec<T>
HashMap<K, V, H>
```

Product type

```
#[derive(Clone, Debug)]
Struct User {
  id: Bytes,
  name: String,
  gender: Gender
}

impl Authenticated for User {}
```

```
enum Gender {
  Unspecified = 0,
  Female = 1,
  Male = 2
}
```

Trait

```
pub trait Authenticated {
  fn is_authenticated(&self);
  fn is_admin(&self);
}

pub trait Future {
    type Output;
    pub fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

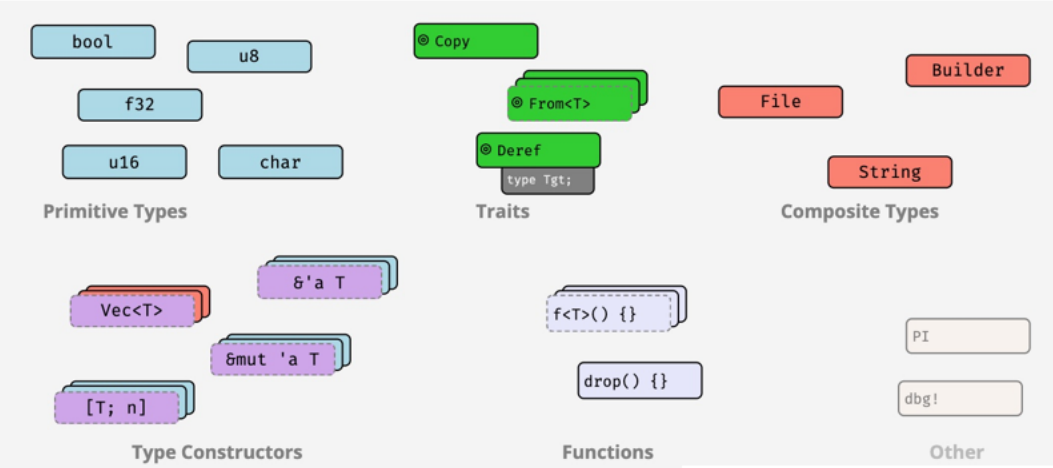
Sum Type

```
enum State {
  Init,
  SynSent(ConnState),
  SynReceived(ConnState),
  Done(Connection),
}
```

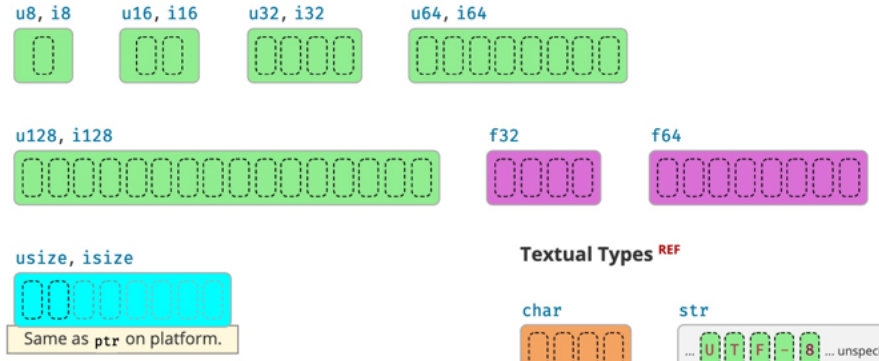
```
struct ConnState {
  ...
}
```

```
Struct Connection {
  ...
}
```

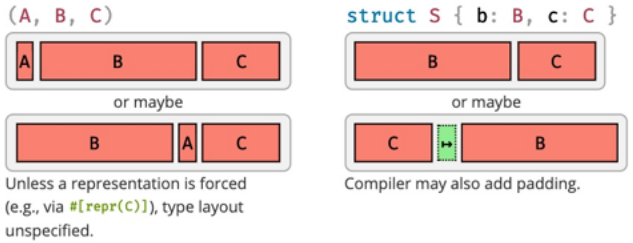
数据结构在内存中是如何排布的?



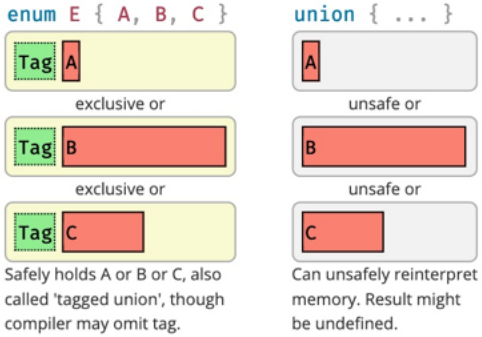
Numeric Types REF

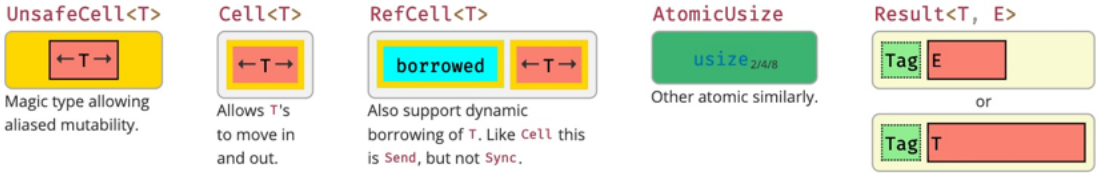


Textual Types REF



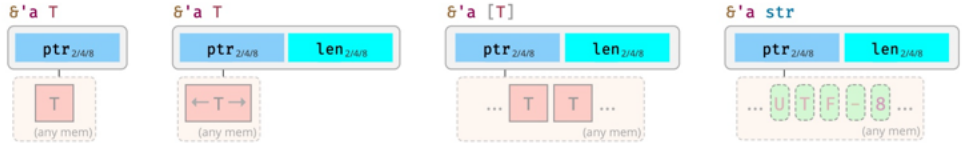
These **sum types** hold a value of one of their sub types:

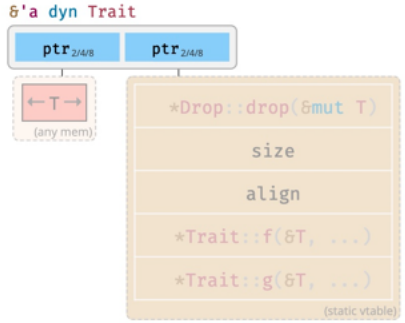




Pointer Meta

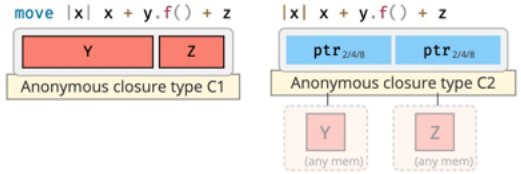
Many reference and pointer types can carry an extra field, **pointer metadata**. STD It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

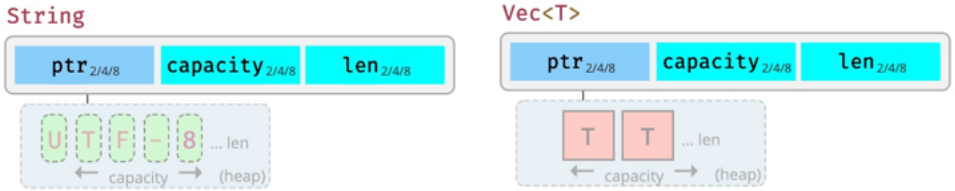




Closures

Ad-hoc functions with an automatically managed data block **capturing** REF environment where closure was defined. For example:





Observe how String differs from &str and &[char].

Trait (Typeclass)

refer to the actual data

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }

    ...
}
```

methods must be implemented

methods have a default impl

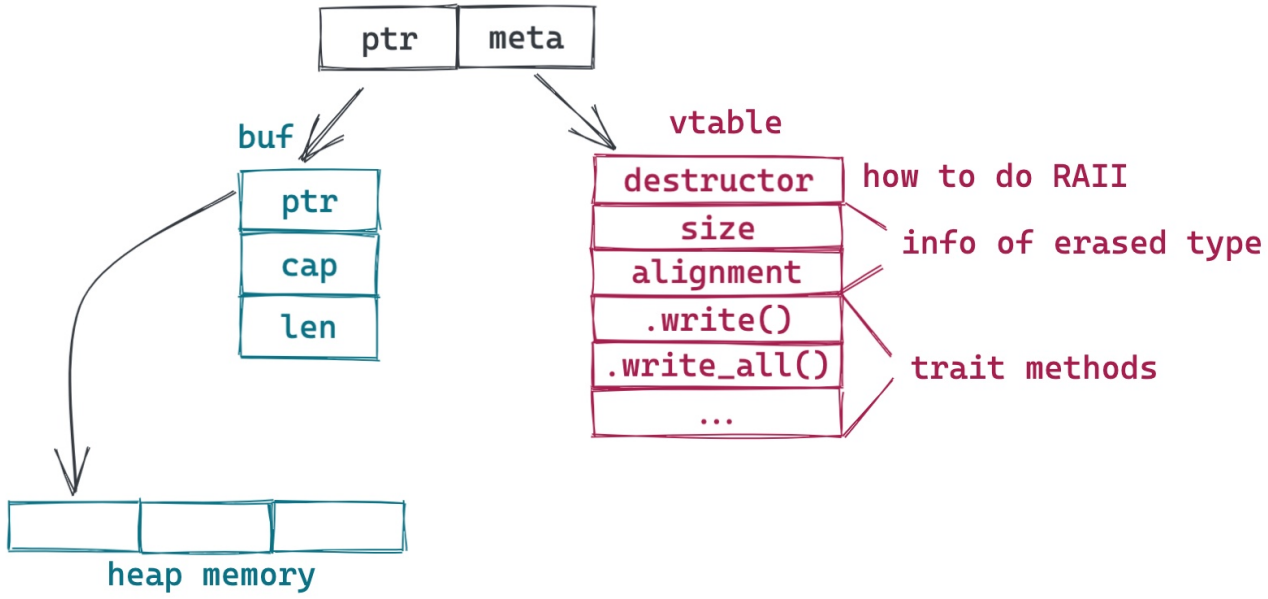
```
struct Sink {
    total: usize,
}

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        self.total += buf.len();
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

Coherence rule: either trait or type must be new in the current crate.

when implementing trait methods, you can access the actual data

Live coding：Fibonacci 遍历器

```rust
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: Write = buf; // `Write` does not have a constant size

let writer: &mut Write = &mut buf; // ok. Trait object
```



Trait Object

- 你无法直接把值赋给 trait
  - 不像 java，Rust 没有隐式引用
- trait object 是胖指针（自动生成）
  - ptr：指向数据的指针
  - vptr：指向 vtable 的指针
- 动态分发

```rust
pub trait Formatter {
    fn format(&self, input: &mut str) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct RustFormatter;
impl Formatter for RustFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

pub fn format(input: &mut str, formatters: Vec<Box<dyn Formatter>>) {
    for formatter in formatters {
        formatter.format(input);
    }
}
```

More about trait

- associated type
- generics
- supertrait
- trait composition

```
pub trait Iterator {
    type Item;
    pub fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

associate type

```
pub trait From<T> {
    pub fn from(T) -> Self;
}
```

generic type

```
trait Person {
    fn name(&self) -> String;
}

trait Student: Person {
    fn university(&self) -> String;
}

trait Programmer {
    fn fav_language(&self) -> String;
}

trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
```

super trait

trait composition

Generics

History of Generic Programming

The first step was a generalized machine architecture, exemplified by the IBM 360, based on a uniform view of the machine memory as a sequence of bytes, referenced by uniform addresses (pointers) independent of the type of data being referenced. The next step was the C programming language [KeRi78], which was effectively a generalized machine language for such architectures, providing composite data types to model objects in memory, and pointers as identifiers for such memory objects with operations (dereferencing and increment/decrement) that were uniform across types. The C++ programming language [Stroustrup97] was the next step. It allows us to generalize the use of C syntax, applying the built-in operators to user types as well, using class definitions, operator overloading, and templates. The final step in this progression is generic programming

(from: Fundamentals of Generic Programming)

Generics 之于 Types

就像

Types 之于 Values

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T>(a: T, b: T) -> T { a + b }
```

```
func add<T>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: t -> t -> t
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

```
pub trait Add<Rhs = Self> {
    type Output;
    pub fn add(self, rhs: Rhs) -> Self::Output;
}
```

SystemTime + Duration => SystemTime

```
func add<T: Numeric>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: Num t => t -> t -> t
add a b = a + b
```

Example	Explanation
`S<T>`	A **generic** BK EX type with a type parameter (`T` is placeholder name here).
`S<T: R>`	Type short hand **trait bound** BK EX specification (`R` *must* be actual trait).
`T: R, P: S`	**Independent trait bounds** (here one for `T` and one for `P`).
`T: R, S`	Compile error, you probably want compound bound `R + S` below.
`T: R + S`	**Compound trait bound** BK EX, `T` must fulfill `R` and `S`.
`T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has lifetimes, must outlive `'a`.
`T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. ?
`T: 'a`	Type **lifetime bound** EX; if T has references, they must outlive `'a`.
`T: 'static`	Same; does esp. *not* mean value `t` *will* live `'static`, only that it could.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`S<const N: usize>`	**Generic const bound**; ? user of type `S` can provide constant value `N`.
`S<10>`	Where used, const bounds can be provided as primitive values.
`S<{5+5}>`	Expressions must be put in curly brackets.
`S<T> where T: R`	Almost same as `S<T: R>` but more pleasant to read for longer bounds.
`S<T> where u8: R<T>`	Also allows you to make conditional statements involving *other* types.
`S<T = R>`	**Default type parameter** BK for associated type.
`S<'_>`	Inferred **anonymous lifetime**; asks compiler to *'figure it out'* if obvious.
`S<_>`	Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()`
`S::<T>`	**Turbofish** STD call site type disambiguation, e.g. `f::<u32>()`.
`trait T<X> {}`	A trait generic over `X`. Can have multiple `impl T for S` (one per `X`).
`trait T { type X; }`	Defines **associated type** BK REF `X`. Only one `impl T for S` possible.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`, here `T` type parameter.
`impl S<T> {}`	Implement functionality for exactly `S<T>`, here `T` specific type (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types**, BK returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**", BK somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch**, BK REF `f` will not be monomorphized.
`fn f() where Self: R;`	In `trait T {}`, make `f` accessible only on types known to also `impl R`.
`fn f() where Self: R {}`	Esp. useful w. default methods (non dflt. would need be impl'ed anyway).
`for<'a>`	**Higher-ranked trait bounds.** NOM REF
`trait T: for<'a> R<'a> {}`	Any `S` that `impl T` would also have to fulfill `R` for any lifetime.

Generic Programming Example

```
int binary_search(int x[], int n, int v) {
  int l = 0;
  int u = n;

  while (true) {
    if (l > u) return -1;

    int m = (l + u) / 2;

    if (x[m] < v) l = m + 1;
    else if (x[m] == v) return m;
    else u = m - 1;
  }
}
```

first ForwardIterator value_type(I), comparable

```
template <class I, class T>
I lower_bound(I f, I l, const T& v) {
  while (f ≠ l) {
    auto m = next(f, distance(f, l) / 2);

    if (*m < v) f = next(m);
    else l = m;
  }
  return f;
}
```

last

demo: rust implmentation

Live coding: Event encoder

Realworld GP example

Underlying protocols: TCP, WebSocket, HTTP/2, QUIC, etc.

```rust
pub struct AsyncProstWriter<W, T, D> {
    writer: W,
    pub(crate) written: usize,
    pub(crate) buffer: Vec<u8>,
    from: PhantomData<T>,
    dest: PhantomData<D>,
}
```

Sink

Stream

```rust
pub struct AsyncProstReader<R, T, D> {
    reader: R,
    pub(crate) buffer: BytesMut,
    into: PhantomData<T>,
    dest: PhantomData<D>,
}
```

```
v1 & v2 shared logic:

- read data based on len
- write data to writer

deferred decision:

- underlying network protocols
- Message types
- decode body or not
```

V1

4 bytes protobuf encoded buf

len | payload

max: 64k

Client

Server

```
message Data {
  oneof value {
    ...
  }
}
```

V2

4 bytes

hlen | len | Option<H> | Body

```rust
pub struct Frame<H, T> {
    pub header: Option<H>,
    pub body: Option<Either<Vec<u8>, T>>,
}
impl<H, T> Framed for Frame<H, T>
where
    H: Message + ShallDecodeBody + Default,
    T: Message + Default
```

if exists, check if need to decode body
otherwise, decode body

```rust
pub trait ShallDecodeBody {
    fn shall_decode_body(&self) -> bool;
}
```

```rust
impl<W, T, D> Sink<T> for AsyncProstWriter<W, T, D>
where
    W: AsyncWrite + Unpin,
    Self: ProstWriterFor<T>,


pub trait ProstWriterFor<T> {
    fn append(&mut self, item: T) -> Result<(), io::Error>;
}

impl<W, F: Framed> ProstWriterFor<F>
for AsyncProstWriter<W, F, AsyncFrameDestination> { ... }

impl<W, T: Message> ProstWriterFor<T>
for AsyncProstWriter<W, T, AsyncDestination> { ... }
```

```rust
impl<R, T> Stream for AsyncProstReader<R, T, AsyncDestination>
where
    T: Message + Default,
    R: AsyncRead + Unpin { ... }


impl<R, T> Stream for AsyncProstReader<R, T, AsyncFrameDestination>
where
    R: AsyncRead + Unpin,
    T: Framed + Default { ... }
```

> Functions delay binding: data structures induce binding.
> Moral: Structure data late in the programming process.
>
> — Epigrams on programming

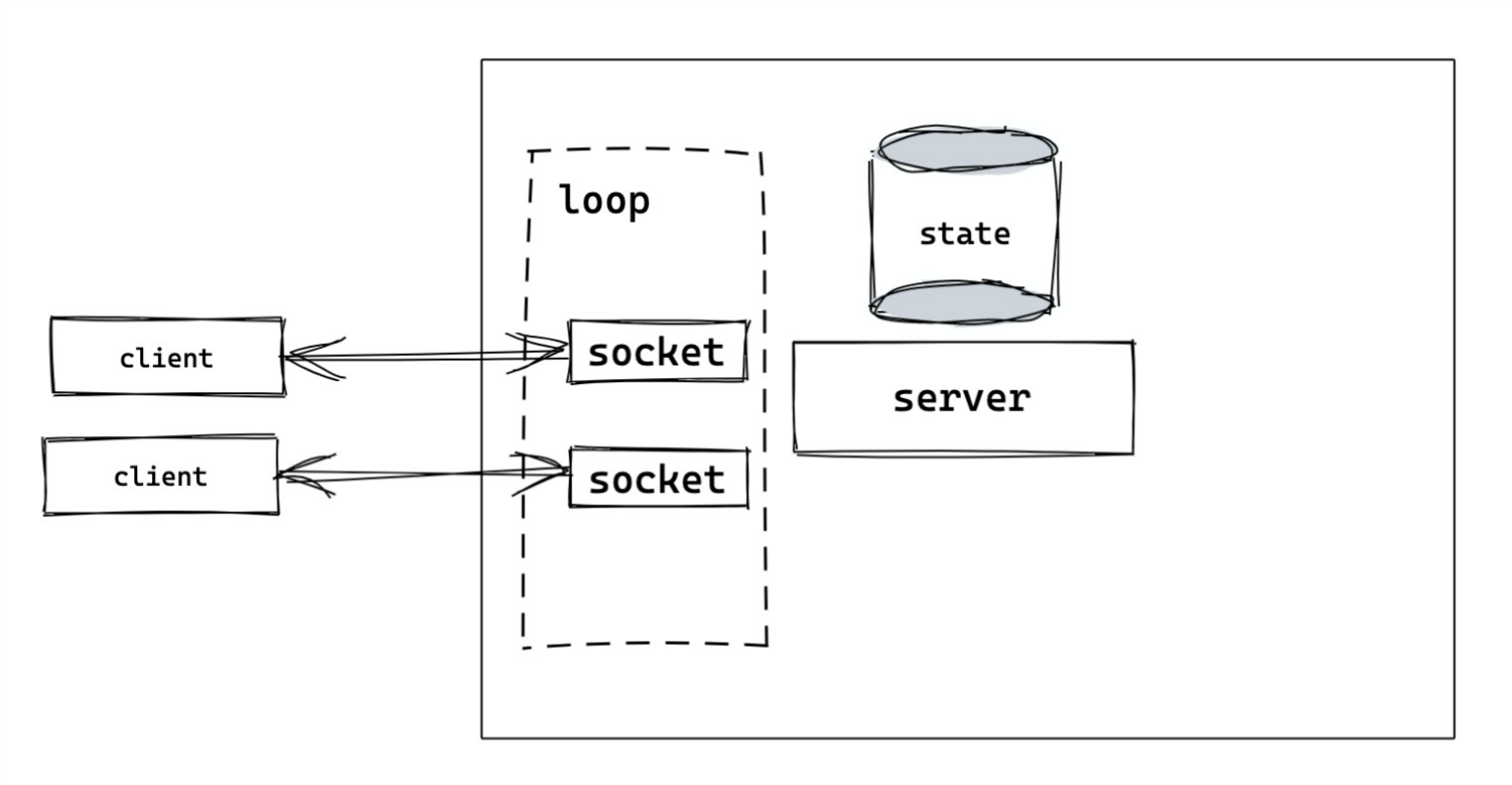
参考文档

- All about trait objects
- Protocol-oriented programming in swift
- Generic Data Types
- Generics (Rust by Example)
- async prost

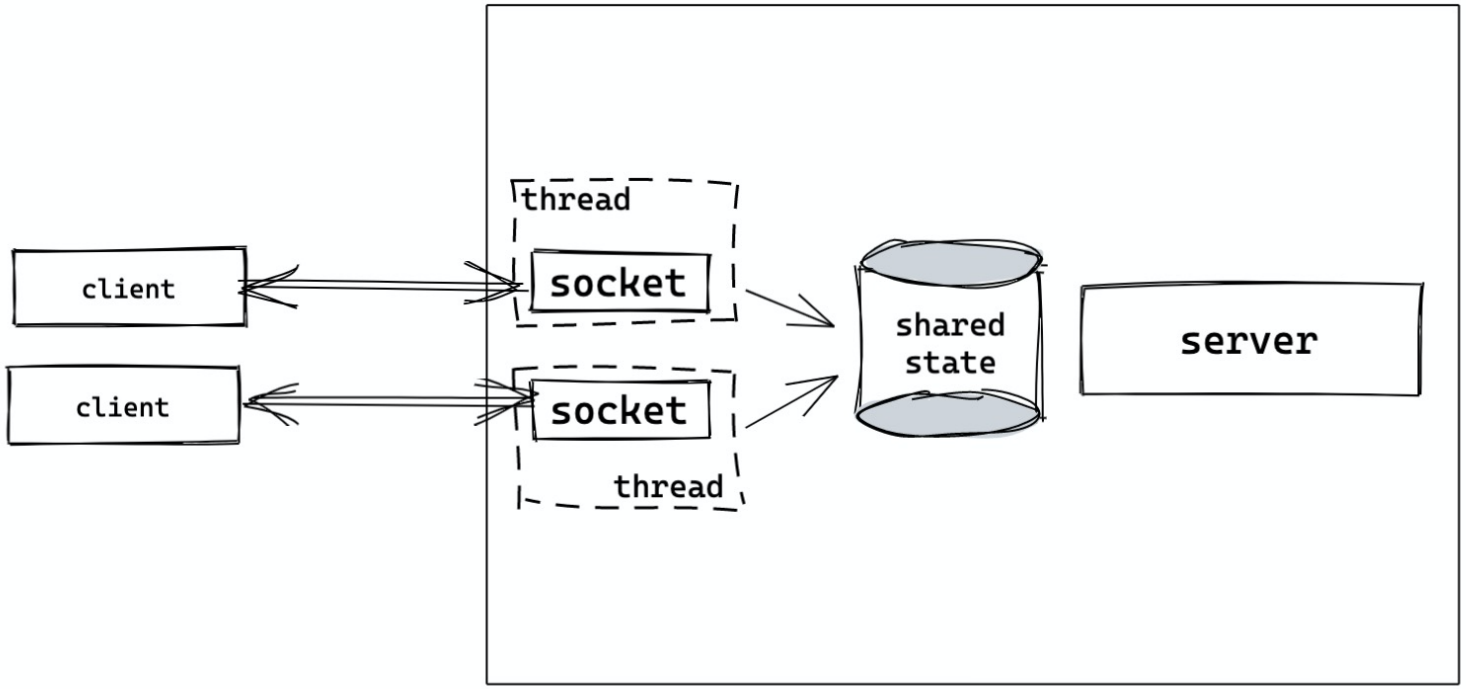
并发 - 并发原语

我们来一起构思一个 KV server

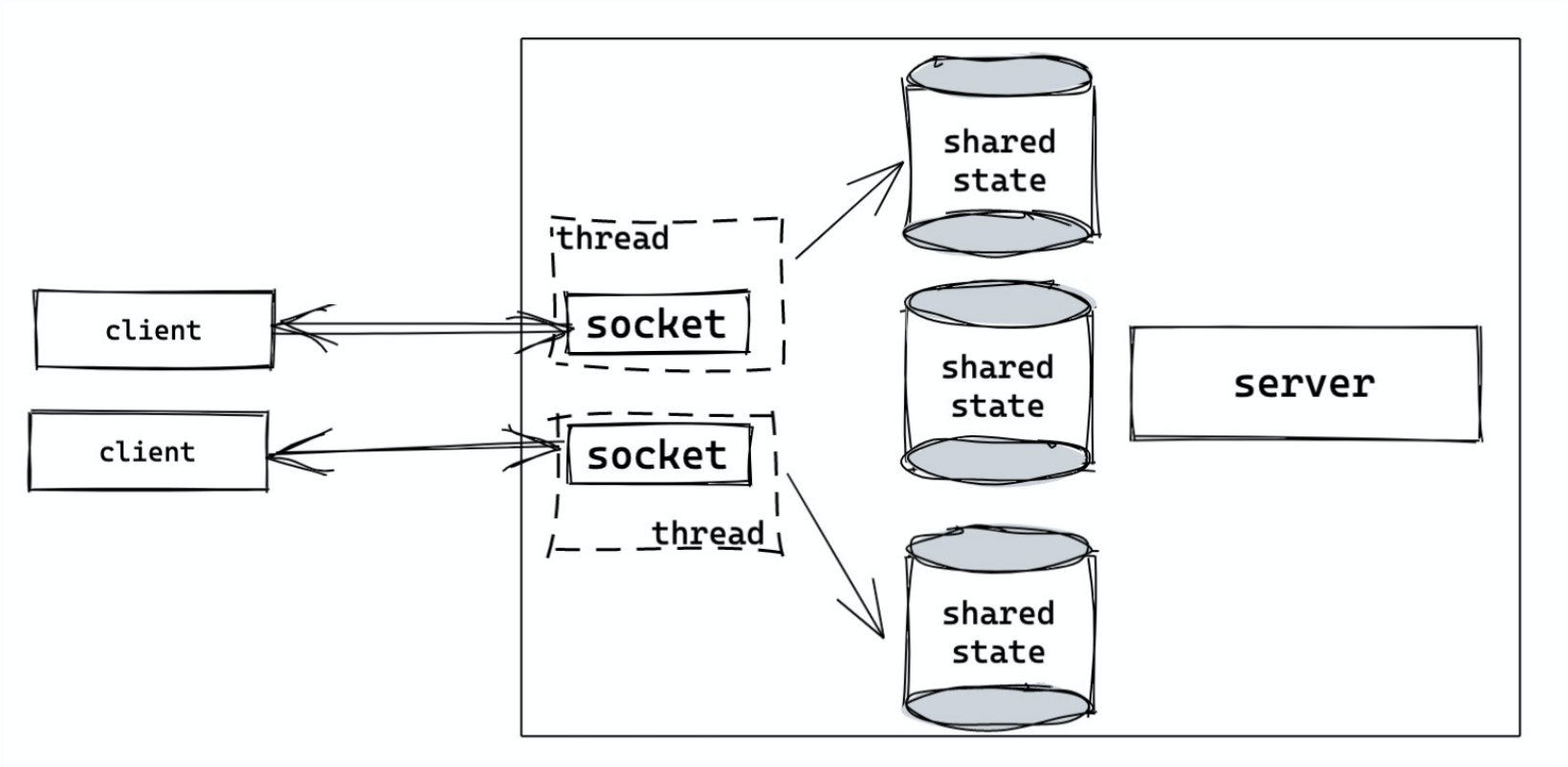
v1: 单线程死循环



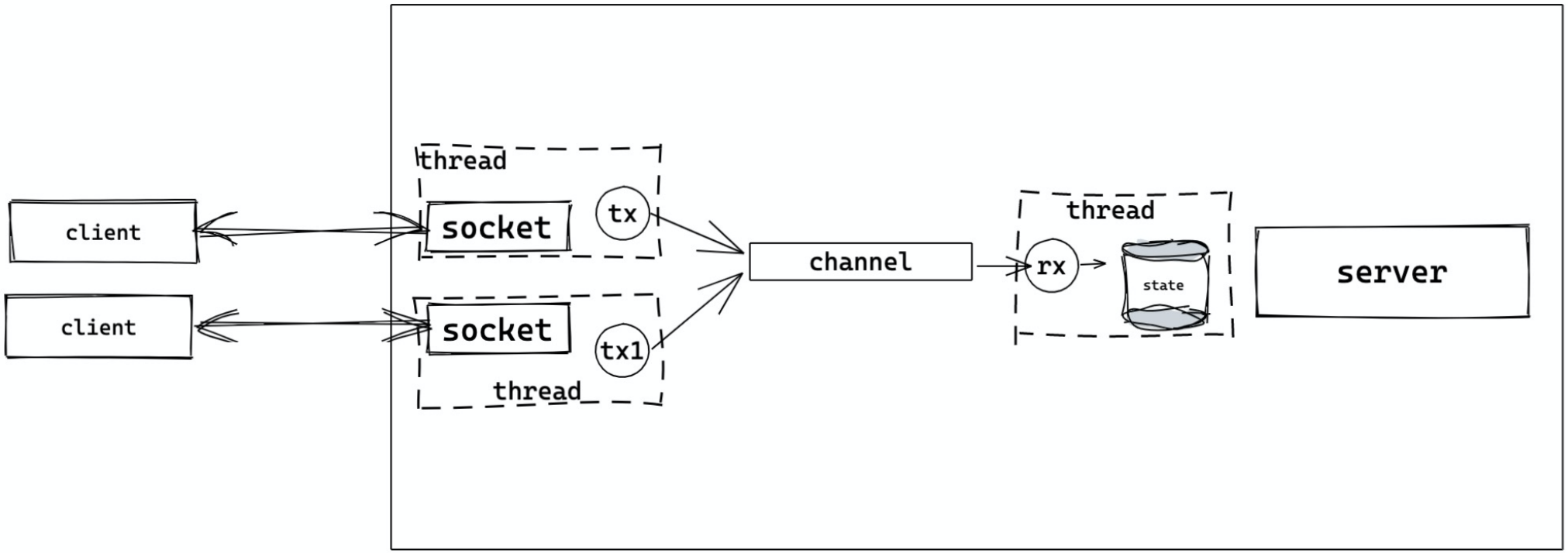
v2: 多线程，共享数据


thread
client
socket
shared
state
server
client
socket
thread

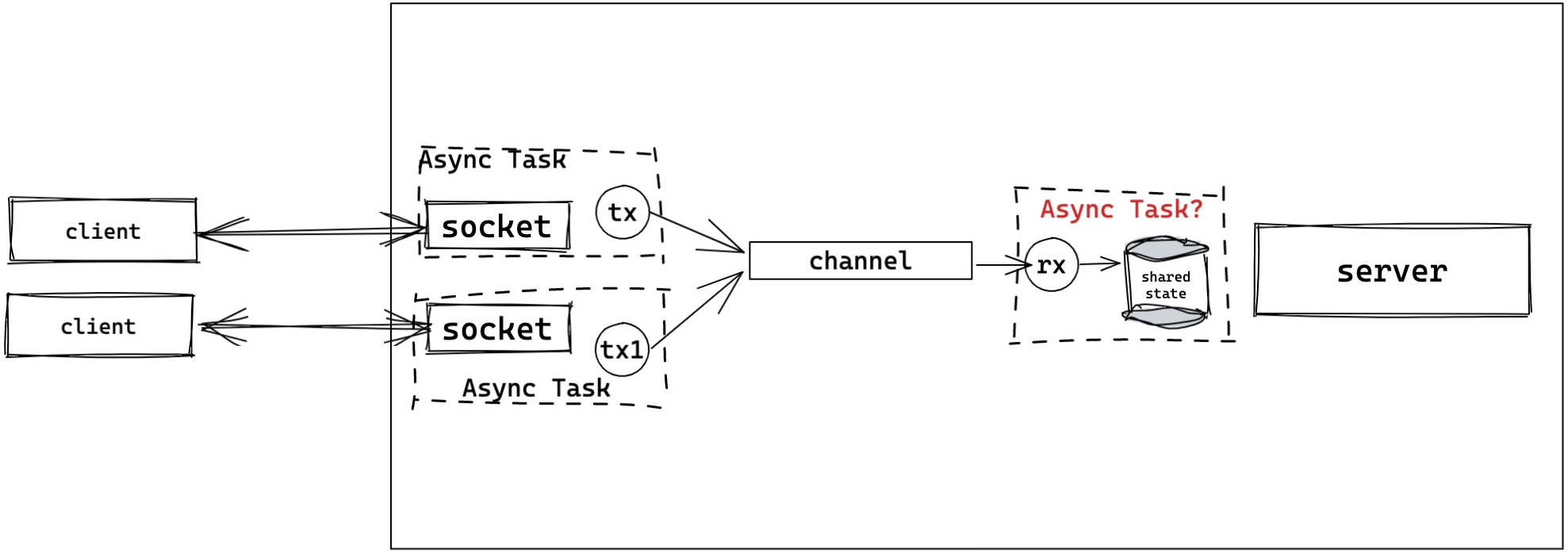
v3: 优化锁的粒度


client
client
thread
socket
socket
thread
shared state
shared state
shared state
server

v4: 使用 Channel



v5: 异步处理


client
client
Async Task
socket
tx
socket
tx1
Async Task
channel
Async Task?
rx
shared state
server

我们目前使用了什么并发原语?

- Mutex Lock
- Channel/Actor
- Async Task (coroutine, promise, etc.)

Mutex 是如何构建的?

一个比较糙的实现

```
struct Lock<T> {
  locked: bool,
  data: T,
}

impl<T> Lock<T> {
  pub fn new(data: T) -> Lock<T> { ... }

  pub fn lock<R>(&mut self, op: impl FnOnce(&mut T) -> R) -> R {
    // spin if we can't get lock
    while self.locked != false {} // **1
    // ha, we can lock and do our job
    self.locked = true; // **2
    // execute the op as we got lock
    let ret = op(self.data); // **3
    // unlock
    self.locked = false; // **4
    ret
  }
}

// You may call it like this:
let l = Lock::new(0);
l.lock(|v| v += 1);
```

问题

- 原子性
  - 在多核环境下，1/2 之间会产生竞争（race condition）- 其它线程也许会进入
  - 在单核环境下，OS 抢占多任务依旧可能会在 1/2 见产生竞争
- 乱序
  - 编译器也许会优化指令，把 3 放在 1 之前
  - CPU 也许会乱序执行（OOO execution），把 3 放在 1 之前

如何解决这个问题?

- 需要硬件（CPU）来保证原子性和避免 OOO
- 算法：**CAS (Compare-And-Swap)**
- 数据结构（Rust）： `AtomicXXX`

Atomics

更新之后的 lock

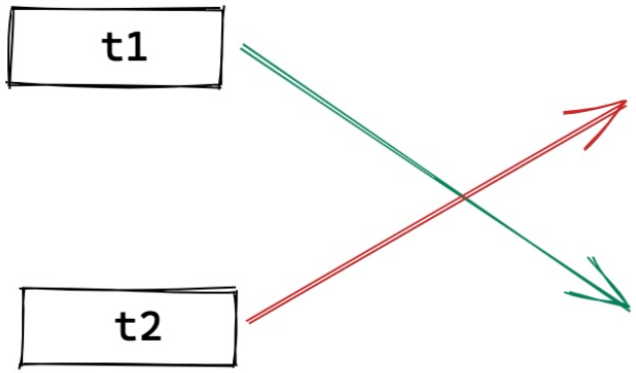
```
struct Lock<T> {
  locked: AtomicBool, // ***
  data: UnsafeCell<T>, // ***
}
unsafe impl<T> Sync for Lock<T> where T: Send {} // need to explicitly impl `Send`

impl<T> Lock<T> {
  pub fn new(data: T) -> Self { ... }
  pub fn lock<R>(&mut self, op: impl FnOnce(&mut T) -> R) -> R {
    // spin if we can't get lock
    while self
      .locked
      .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
      .is_error() {}
    // execute the op as we got lock
    let ret = op(unsafe { &mut *self.v.get() }); // **3
    // unlock
    self.locked.store(false, Ordering::Release); // **4
    ret
  }
}

// You may call it like this:
let l = Lock::new(0);
l.lock(|v| v += 1);
```

`Ordering` 是什么概念?

- Relaxed: 没有限制，随意 OOO
- Release:
  - 对当前线程，任何 read/write 不能被乱序到这条指令之后（比如：`store`）
  - 对于其它线程，如果用 `Acquire` read，会看到变化后的结果
- Acquire:
  - 对当前线程，任何 read/write 不能被乱序到这条指令之后（比如：`compare_exchange`）
  - 对于其它线程，如果用 `Release` 来更新数据，更新的数据会被当前线程看到
- AcqRel: Acquire 和 Release 的结合
- SeqCst: `AcqRel` 之外，所有线程都看到相同的操作顺序

继续优化我们的锁

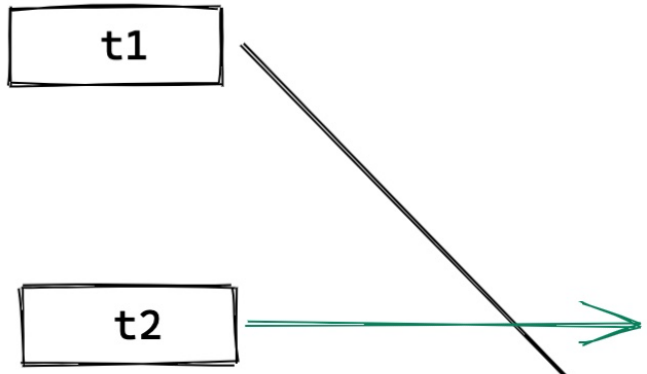
```
pub struct Lock<T> {
    locked: AtomicBool,
    data: UnsafeCell<T>,
}
unsafe impl<T> Sync for Lock<T> where T: Send {}

impl<T> Lock<T> {
  pub fn new(data: T) -> Self {...}
  pub fn lock<R>(&self, op: impl FnOnce(&mut T) -> R) -> R {
      while self
          .locked
          .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
          .is_err()
      {
          while self.locked.load(Ordering::Relaxed) == true {
              std::thread::yield_now(); // we may yield thread now
          }
      }
      let ret = op(unsafe { &mut *self.data.get() });
      self.locked.store(false, Ordering::Release);
      ret
  }
}
```




```
pub fn with_lock<R>(&self, f: impl FnOnce(&mut T) -> R) -> R {
    while self
            .locked
            .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
            .is_err()
    {
        while self.locked.load(Ordering::Relaxed) == true {}
    }
    let ret = op(unsafe { &mut *self.v.get() });
    self.locked.store(false, Ordering::Release);
    ret
}
```

t1 finished op, released lock

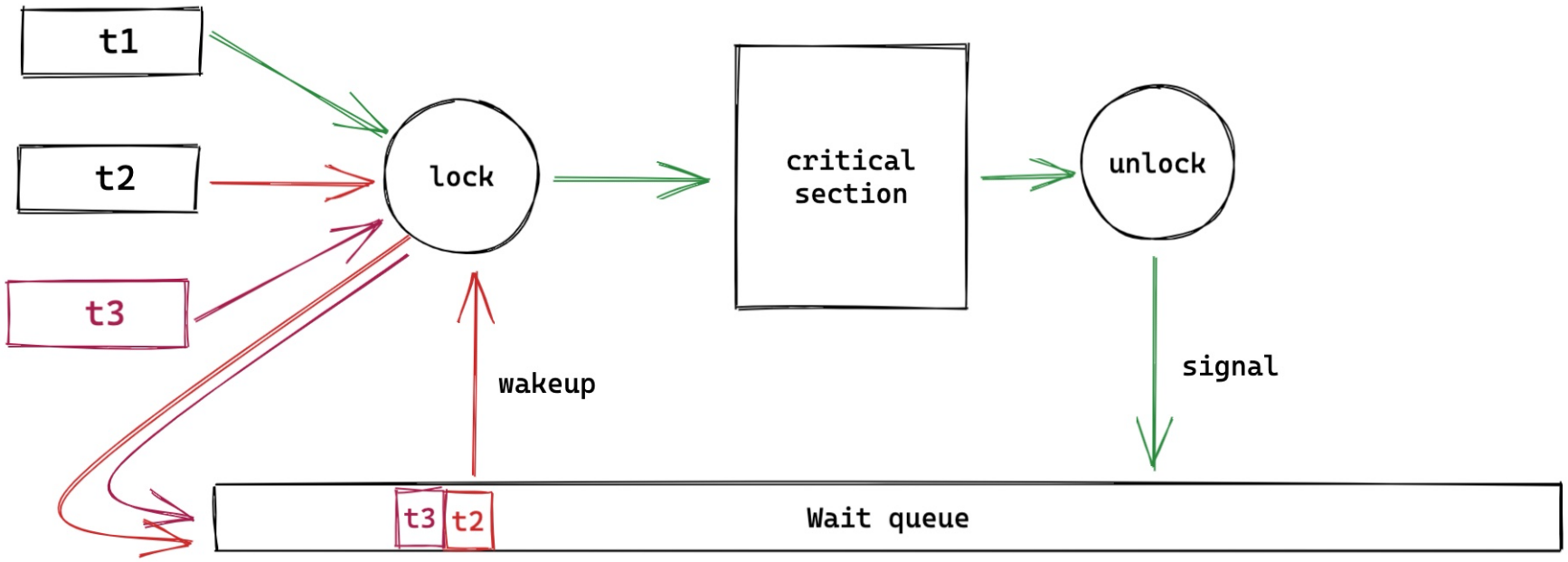
t2 finished spin, entered critical section




```
pub fn with_lock<R>(&self, f: impl FnOnce(&mut T) -> R) -> R {
    while self
            .locked
            .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
            .is_err()
    {
        while self.locked.load(Ordering::Relaxed) == true {}
    }
    let ret = op(unsafe { &mut *self.v.get() });
    self.locked.store(false, Ordering::Release);
    ret
}
```

这是 Mutex 如何运作的基础

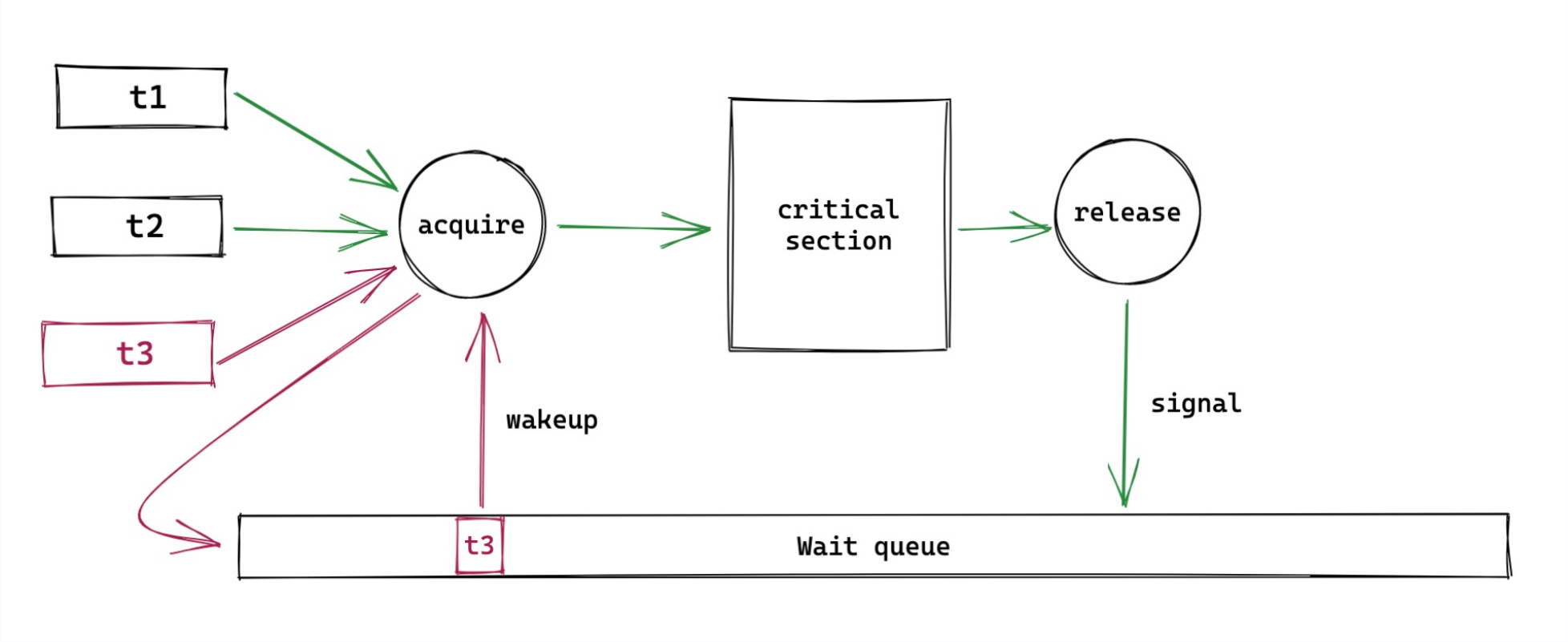
Real world Mutex



Semaphore

> In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.
>
> A useful way to think of a semaphore as used in a real-world system is as **a record of how many units of a particular resource are available**, coupled with operations to adjust that record safely (i.e., to avoid race conditions) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.

Semaphore: 更一般化的 Mutex

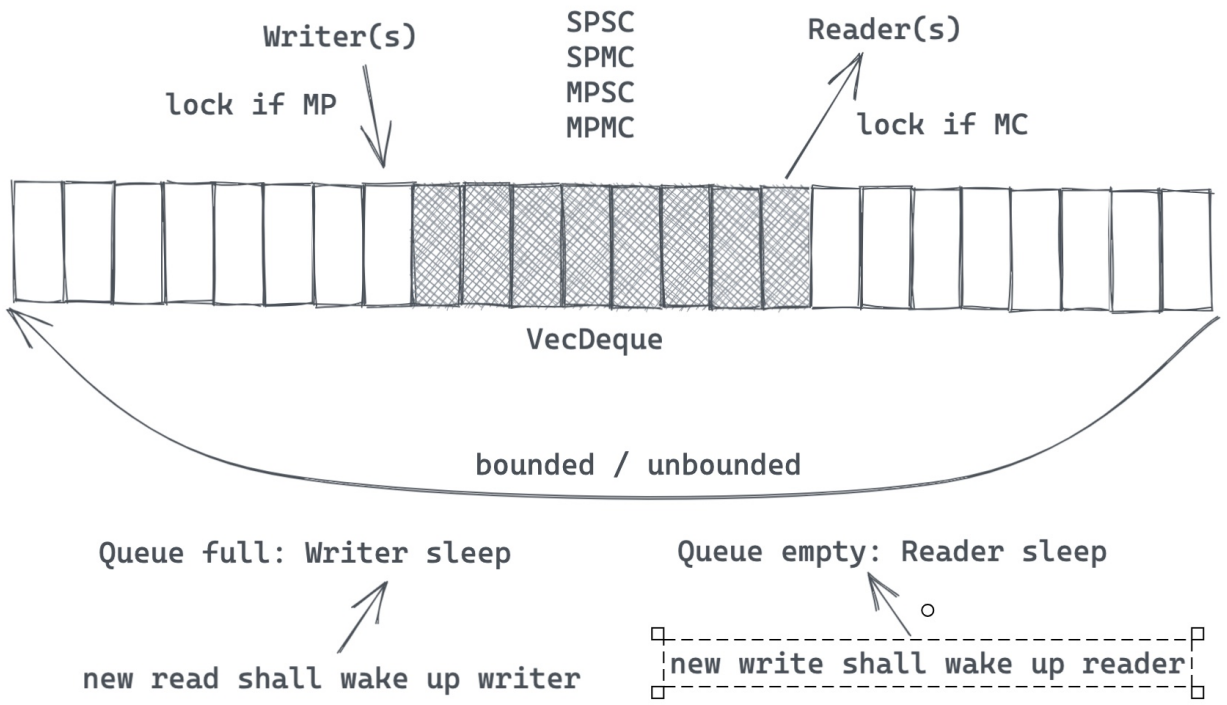


Demo code: restricted HTTP client

code/primitives/src/http_semaphore.rs

Channel

Channel 基础


Writer(s)
lock if MP
SPSC
SPMC
MPSC
MPMC
Reader(s)
lock if MC
VecDeque
bounded / unbounded
Queue full: Writer sleep
new read shall wake up writer
Queue empty: Reader sleep
new write shall wake up reader

各种 Channel 的实现

- sync: 容量有限，发送者会被 block
  - `Mutex` + `Condvar` + `VecDeque`
  - Atomic VecDeque (atomic queue) + `thread::park` + `thread::notify`
- async: 容量无限，发送者不会被 block
  - `Mutex` + `Condvar` + `VecDeque`
  - `Mutex` + `Condvar` + `DoubleLinkedList`
- rendezvous: 容量为 0，用于线程间同步
  - `Mutex` + `Condvar`
- oneshot: 只允许发送一次数据 e.g. Ctrl+C to stop all threads
  - atomic swap
- async/await
  - 和 sync channel 类似，但 `Waker` 不同

Demo code: naive MPSC

code/primitives/src/channel.rs

Live coding: naive actor

- 我们用什么 channel 实现 actor？SPSC, SPMC, MPSC?
- 创建 actor 时，`pid` 是什么?
- 给 actor 发送一个 message 后，actor 如何回复给发送者（`handle_call`）？

参考资料

- CAS: https://en.wikipedia.org/wiki/Compare-and-swap
- Ordering: https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html
- std::memory_order: https://en.cppreference.com/w/cpp/atomic/memory_order
- Atomics and Memory Ordering: https://www.youtube.com/watch?v=rMGWeSjctlY
- spinlock: https://en.wikipedia.org/wiki/Spinlock
- spin-rs: https://github.com/mvdnes/spin-rs
- parking lot： https://github.com/Amanieu/parking_lot
- Flume: https://github.com/zesterer/flume
- Crossbeam channel： https://docs.rs/crossbeam-channel

并发 - async/await

在 Rust 中使用线程

```rust
use std::thread;

fn main() {
    println!("So we start the program here!");
    let t1 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(200));
        println!("We create tasks which gets run when they're finished!");
    });

    let t2 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(100));
        println!("We can even chain callbacks...");
        let t3 = thread::spawn(move || {
            thread::sleep(std::time::Duration::from_millis(50));
            println!("...like this!");
        });
        t3.join().unwrap();
    });
    println!("While our tasks are executing we can do other stuff here.");

    t1.join().unwrap();
    t2.join().unwrap();
}
```

线程的缺点

- 调用栈太大（不适合大量高并发任务 - 如 web server）
- 上下文切换不受控制
- 上下文切换效率不高（尤其大量线程的环境）

有什么替代方案?

Green threads/processes

(stackful coroutine)

Golang/Erlang

Green Threads

- Run some non-blocking code.
- Make a blocking call to some external resource.
- CPU "jumps" to the "main" thread which schedules a different thread to run and "jumps" to that stack.
- Run some non-blocking code on the new thread until a new blocking call or the task is finished.
- CPU "jumps" back to the "main" thread, schedules a new thread which is ready to make progress, and "jumps" to that thread.

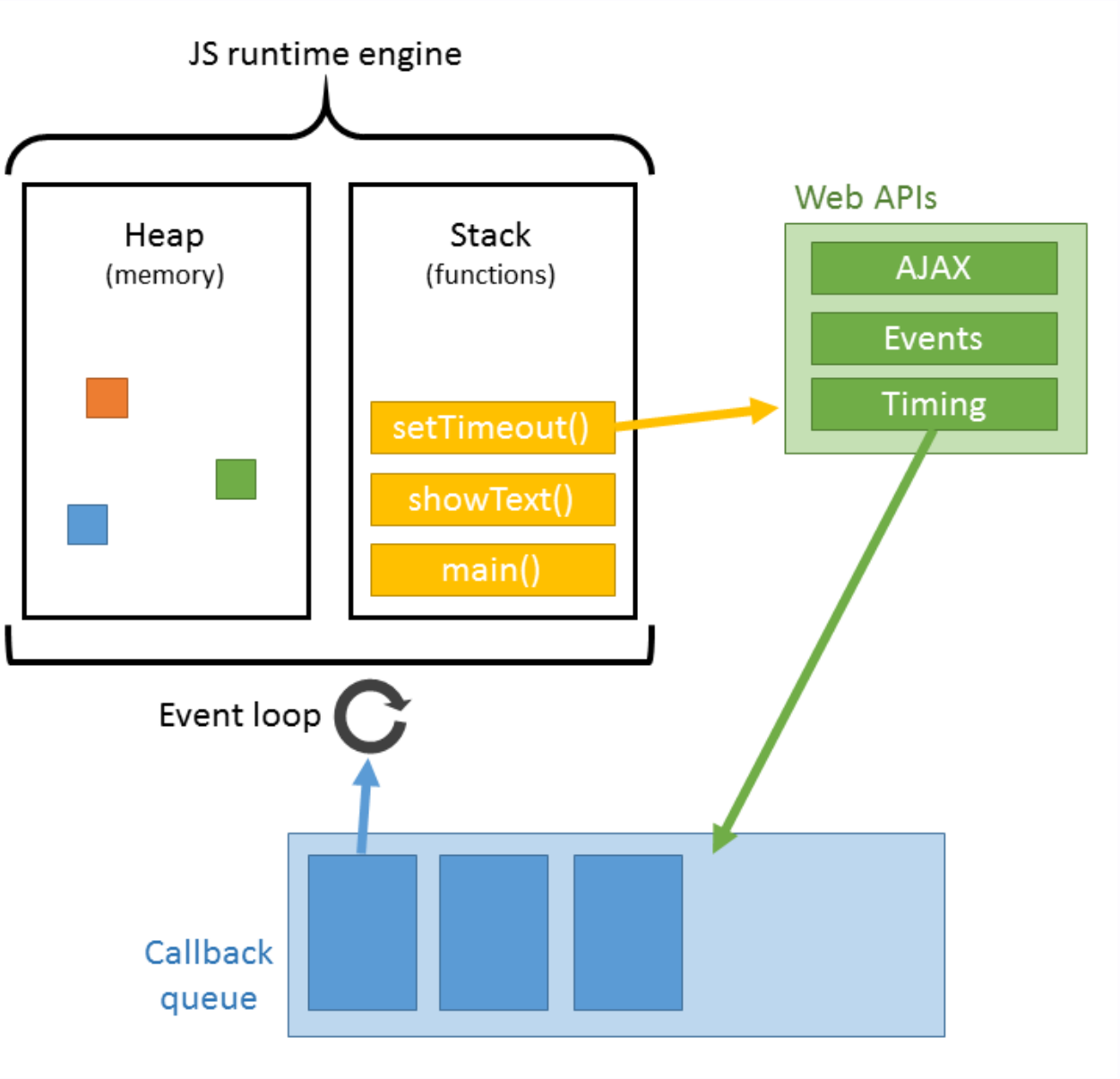
Green Threads - pros/cons

- Pros:
  - Simple to use. The code will look like it does when using OS threads.
  - A "context switch" is reasonably fast.
  - Each stack only gets a little memory to start with so you can have hundreds of thousands of green threads running.
  - It's easy to incorporate preemption which puts a lot of control in the hands of the runtime implementors.
- Cons:
  - The stacks might need to grow. Solving this is not easy and will have a cost.
  - You need to save the CPU state on every switch.
  - It's not a zero cost abstraction (Rust had green threads early on and this was one of the reasons they were removed).
  - Complicated to implement correctly if you want to support many different platforms.

Poll based event loops

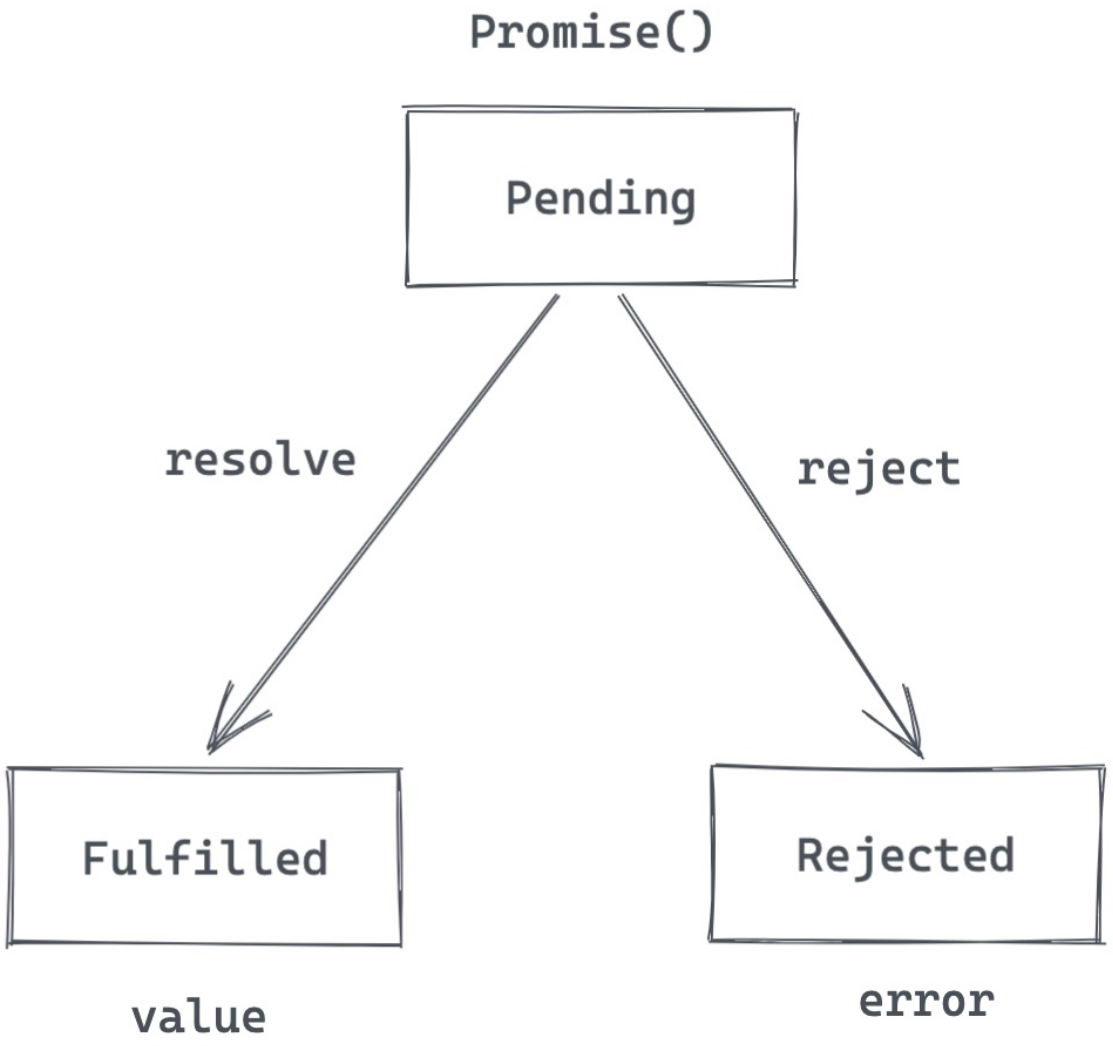
(stackless coroutine)

Javascript/Rust



Callback

```
setTimer(200, () => {
  setTimer(100, () => {
    setTimer(50, () => {
      console.log("I'm the last one");
    });
  });
});
```



Promise

```
function timer(ms) {
    return new Promise(
      (resolve) => setTimeout(resolve, ms)
    );
}

timer(200)
.then(() => timer(100))
.then(() => timer(50))
.then(() => console.log("I'm the last one"));
```

Async/Await

```
async function run() {
    await timer(200);
    await timer(100);
    await timer(50);
    console.log("I'm the last one");
}
```

The Rust approach

```
fn step1<'a>(xZ: &'a u8) -> impl Future<Output=u8> + 'a {}
```

syntax sugar to

```
async fn step1(x: &u8) -> u8 {}
async fn do_tasks() {
  step1().await;
  let step2 = async {
    ...
  };
  step2.await;
}
```

```
pub trait Future {
    type Output;
    pub fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```
pub struct Waker {
    waker: RawWaker,
}

pub struct RawWaker {
    data: *const (),
    vtable: &'static RawWakerVTable,
}
```

```
pub struct Context<'a> {
    waker: &'a Waker,
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}
```

Internally generated code
(Zero Cost Abstraction)

illustrated

```
struct DoTasks {
  step1: FutStep1,
  step2: FutStep2,
  state: State,
}

enum State {
  AwaitingStep1,
  AwaitingStep2,
  Done,
}
```

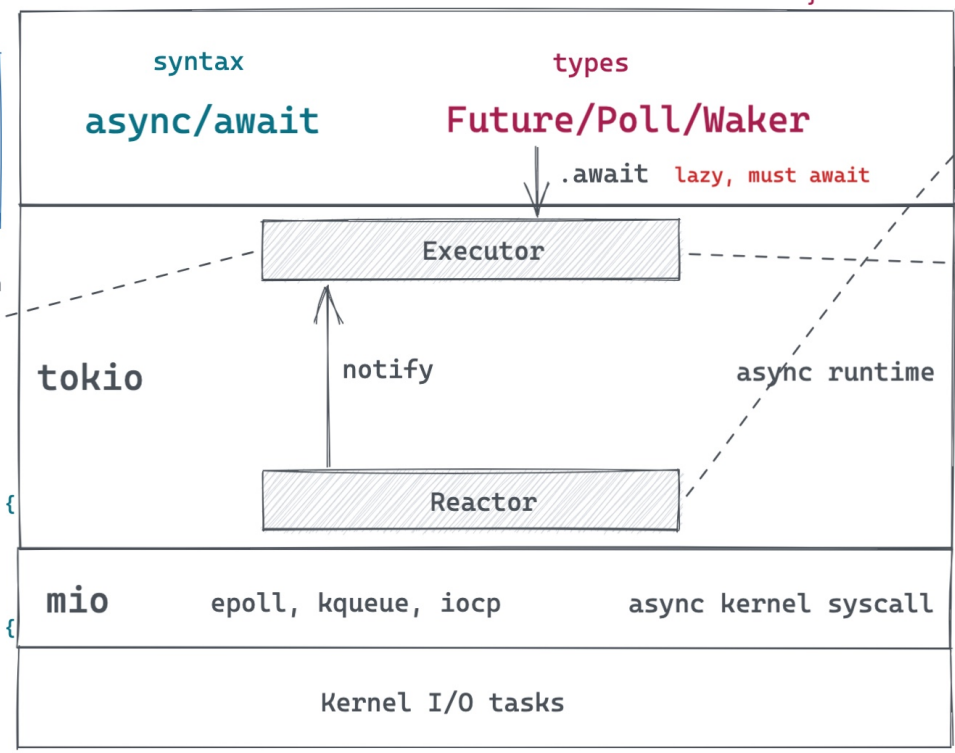
optimized

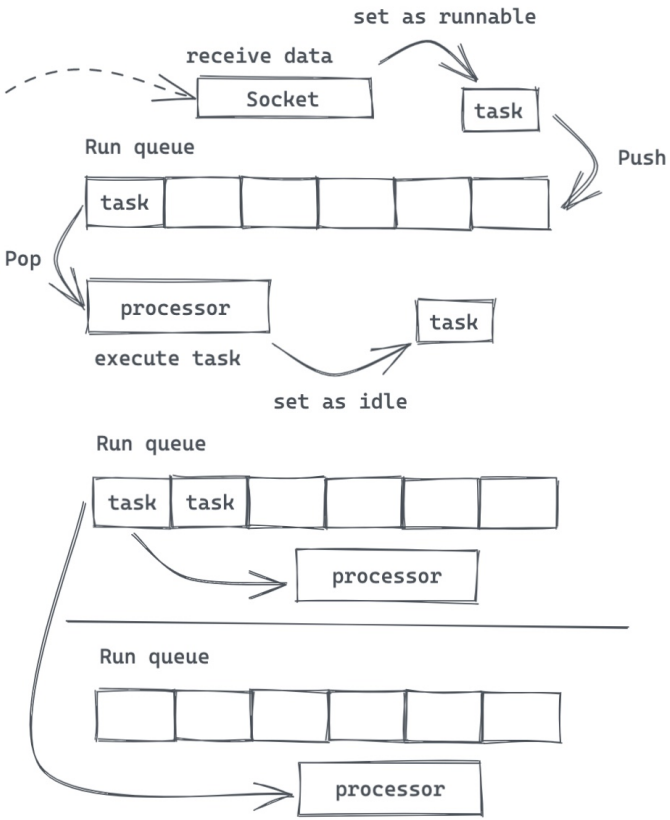
actual

```
enum DoTasks {
  Init,
  AwaitingStep1(FutStep1),
  AwaitingStep2(FutStep2),
  Completed,
}
```

poll future until return

```
impl Future for DoTasks {
  type Output = ();

  fn poll(mut self: Pin<&mut Self>,
       cx: &mut Context<'_>) -> Poll<()> {
    loop {
      match self.state {
        State::AwaitingStep1 => match self.step1.poll(..) {
          Poll::Ready(()) =>
            self.state = State::AwaitingStep2,
          Poll::Pending => return Poll::Pending,
        },
        State::AwaitingStep2 => match self.step2.poll(..) {
        Poll::Ready(()) => self.state = State::Done,
          Poll::Pending => return Poll::Pending,
        },
        State::Done => return Poll::Ready(()),
      }
    }
  }
}
```





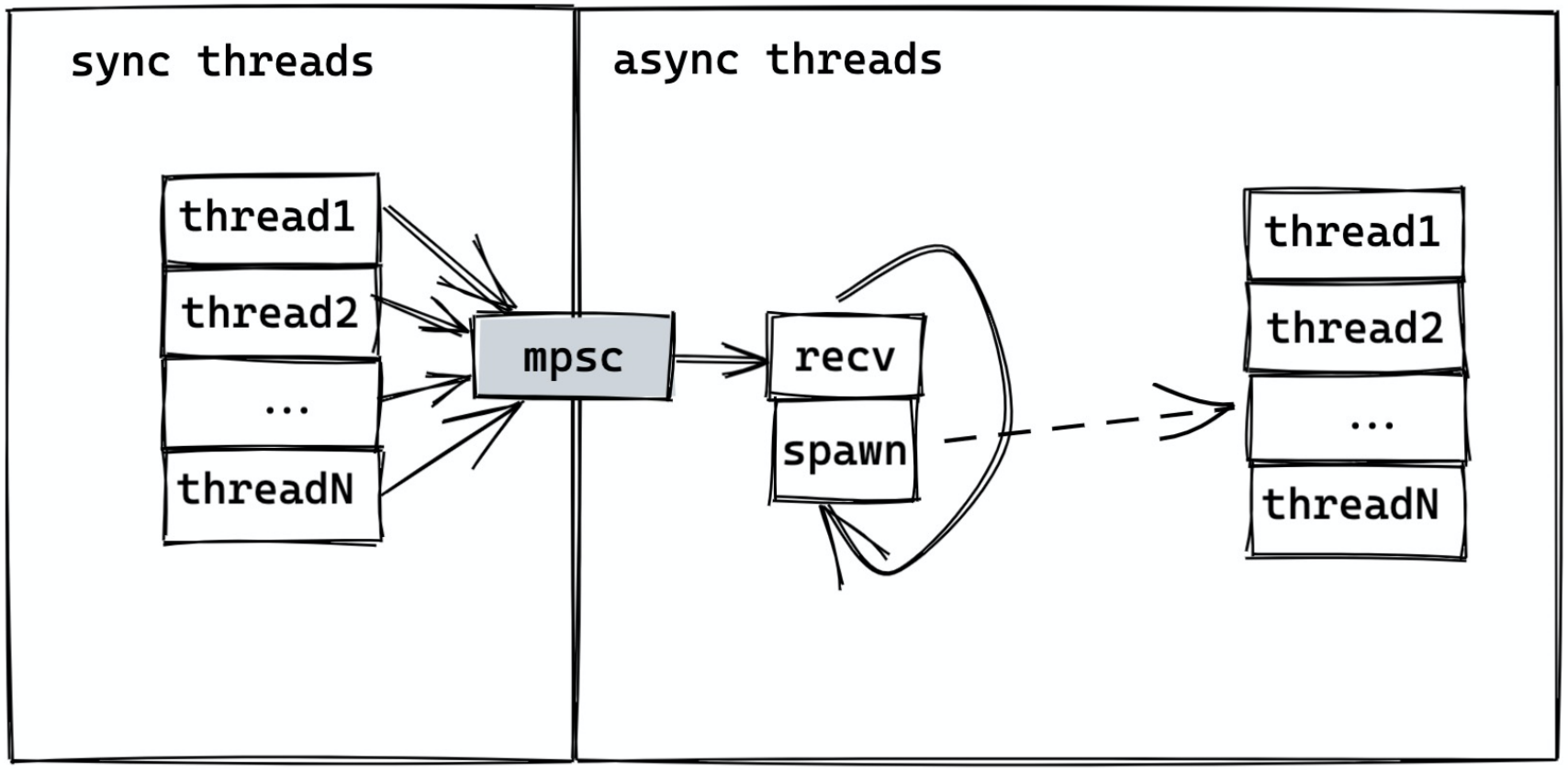
Work stealing scheduler

What is Pin<P>?

a Pin<P> ensures that the pointee of any pointer type P has a stable location in memory, meaning it cannot be moved elsewhere and its memory cannot be deallocated until it gets dropped. We say that the pointee is “pinned”.

Live coding: kv store

One more thing...



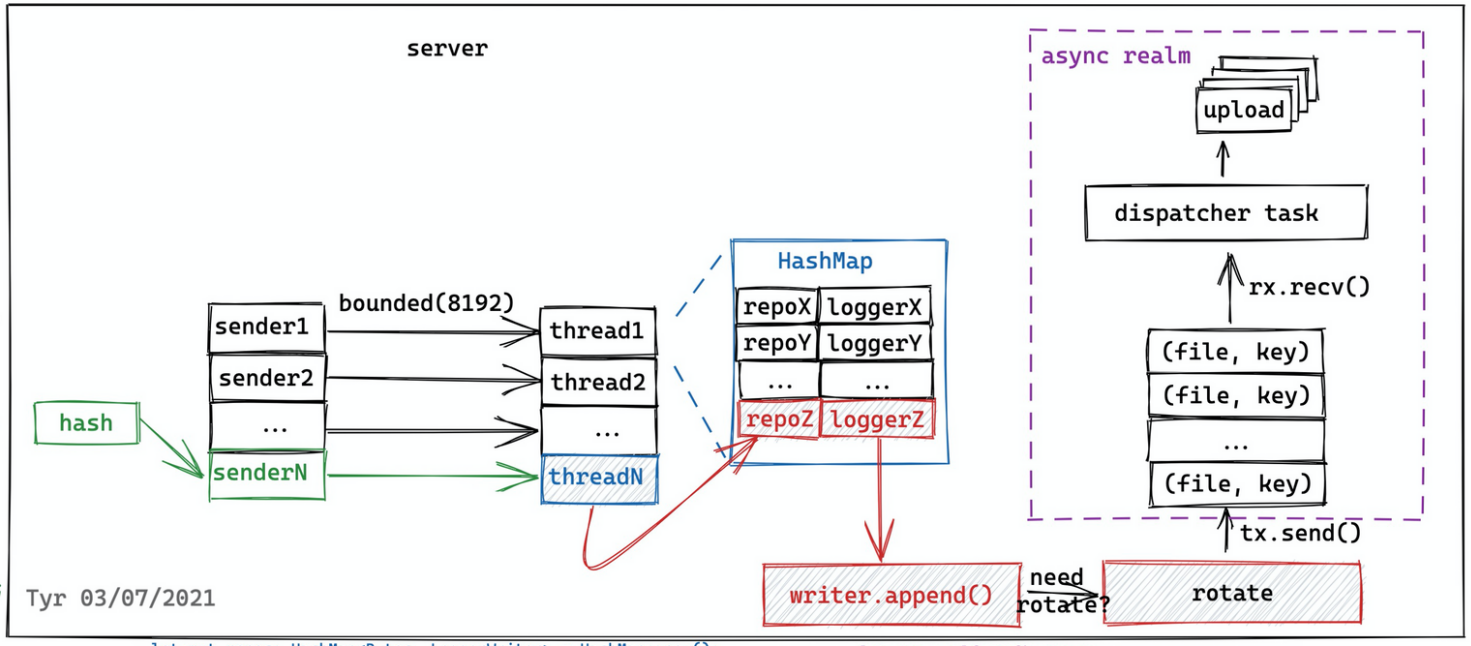
Event store 例子

```
pub fn start(&mut self) {
    for _ in 0..self.config.nthreads {
        let (tx, rx) = bounded(QUEUE_SIZE);
        let config = self.config.clone();
        thread::spawn(move || {
            // see code in blue
        });
        self.senders.push(tx);
    }
}
```

start()

dispatch(repo_id)

```
pub fn dispatch(&self, entry: LogEntry<T>) {
    let mut hasher = AHasher::default();
    entry.repo_id.hash(&mut hasher);
    let thread = hasher.finish() as usize %
            self.config.nthreads as usize;
    self.senders[thread].send(entry).unwrap();
}
```




```
let mut repos: HashMap<Bytes, LoggerWriter> = HashMap::new();

loop {
    let entry: LogEntry<T> = rx.recv().unwrap();
    let repo_id = entry.repo_id.to_vec();
    let writer = repos
        .entry(entry.repo_id)
        .or_insert_with(|| -> LoggerWriter {
            LoggerWriter::new(
                &repo_id,
                &config,
                Some(uploader_tx.clone())
            ).unwrap()
        });
    writer.append(&entry.msg).unwrap();
}
```

```
let cap = self.config.queue;
let (uploader_tx, mut uploader_rx) = mpsc::channel(cap as usize * 32);
let bucket = self.config.backup.bucket.clone();

let client = self.s3_client.clone();
self.rt.spawn(async move {
    loop {
        if let Some((filename, key)) = uploader_rx.recv().await {
            let bucket = bucket.clone();
            let client = client.clone();
            tokio::spawn(async move {
                let uploader = S3Writer::new(client.clone(), bucket);
                uploader.put_object(filename, key).await.unwrap();
            });
        }
    }
});
```

参考文档

- Future explained
- Rust async book
- calloop: a callback based event loop

回顾第一讲

Copy & Move

```
let a = 10;
let b = vec![1, 2, 3];
{
  let (x, y) = (a, b);
}
let c = a; //
let d = b; //
```

Drop

```
{
  let s1 = "Hello world".to_string();
  let arr = vec![Box::new(s1)];
}
```

当 `Drop` 发生时，谁先被 drop？(lifetime 短的)

Sized / DST

- Sized: 编译期 - 可以放在栈上
  - Rust 有 `Sized` trait 和 `?Sized` trait bound
  - 所有的泛型结构默认 `T: Sized`
- DST: 运行期 - 一般只能放在堆上(stack_dst)
  - trait object
  - [T] / str
  - struct / tuple 包含 DST（如：`Mutex`）
  - 对于 DST，指针是 "fat pointer"
- 问题：
  - Q1：指向 slice 的指针在堆还是栈上?
  - Q2：很多数据结构，如 Mutex 包含 DST，为什么它能放在栈上?

Rust 编程范式：实现基本 trait

- Debug/Default/Clone
- Send/Sync/Unpin（一般自动实现了）
- PartialEq/PartialOrd/Hash/Eq/Ord
- Serialize/Deserialize (use feature flag)
- Iterator
- Deref / AsRef / From> / Into

Rust Test 是怎么工作的?

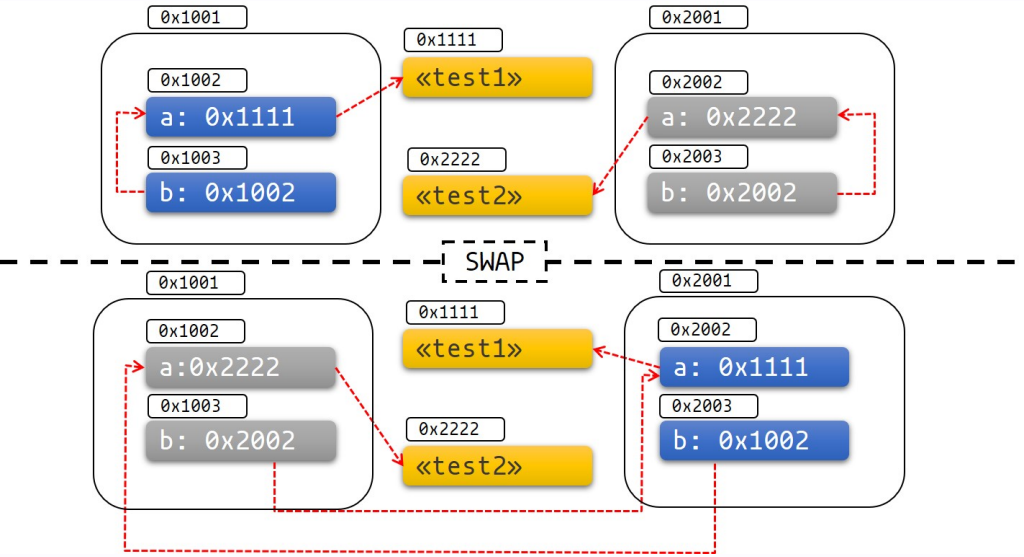
- unit test
- doctest
- integration test：见 tonic
- fuzz：`honggfuzz-rs`（见 snow）
- proptest：`quickcheck` 或 `proptest` （见：cellar）
- benchmark：`[benchmark]` 现在还是 unstable，可以用 `criterion` （见：cellar）

Future: Leaf / Non-leaf

- Leaf：一般是运行时定义的内部事件（e.g. event）/ 外部事件（e.g. io）
  - `tokio::time::Sleep`
  - `tokio::net::TcpStream`
- non-leaf：一般用 async 定义的 future
  - `async move {}`

```
let fut = async {
  let mut stream = TcpStream::connect("localhost:8000").await?;
  stream.write(b"hello world\n").await?;
}
```

Pin



- 自我引用结构在 Rust 下的问题
  - 移动时会出现指针指向的问题
- `Pin<P>` 作用的类型是指针，只要 `P` 没实现 `Unpin`，类型系统可以保证 T 不会被移动
  - 如果 `P` 是 `!Unpin`，`Pin<P>` 无法 `deref_mut`

```
impl<P: DerefMut<Target: Unpin>> DerefMut for Pin<P> {
    fn deref_mut(&mut self) -> &mut P::Target {
        Pin::get_mut(Pin::as_mut(self))
    }
}
```

- `Unpin` 是个 auto trait，编译器会自动实现
  - 但 `impl Future` 是 `!Unpin`
- pin_project

(tokio) AsyncRead / AsyncWrite

```rust
pub trait AsyncRead {
    fn poll_read(self: Pin<&mut Self>, cx: &mut Context<'_>, buf: &mut ReadBuf<'_>)
      -> Poll<Result<()>>;
}

pub trait AsyncWrite {
    fn poll_write(self: Pin<&mut Self>, cx: &mut Context<'_>, buf: &[u8])
      -> Poll<Result<usize, Error>>;
    fn poll_flush(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Result<(), Error>>;
}
```

AsyncRead / AsyncWrite

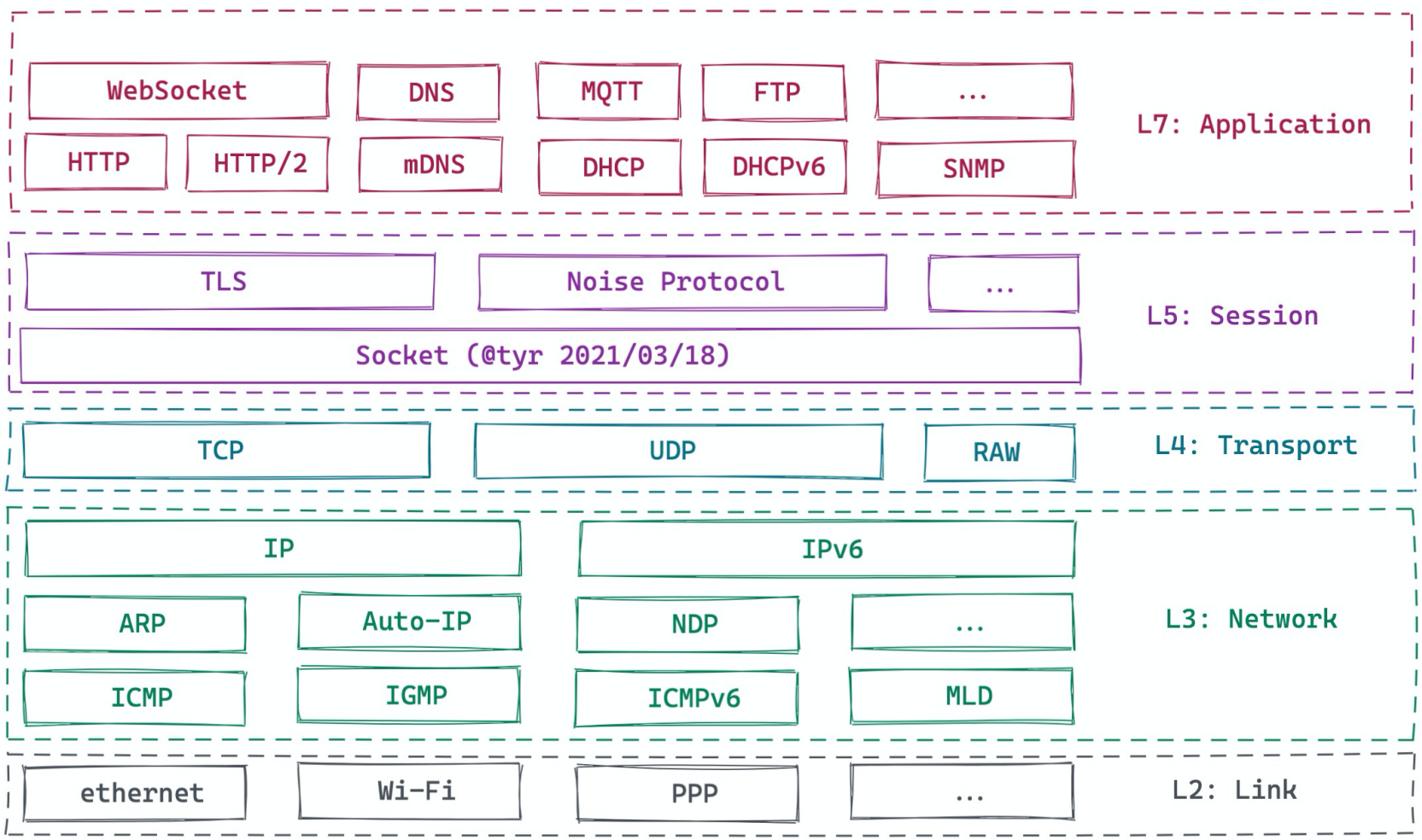
(futures) Stream / Sink

```rust
#[must_use = "streams do nothing unless polled"]
pub trait Stream {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>>;
}

#[must_use = "sinks do nothing unless polled"]
pub trait Sink<Item> {
    type Error;
    fn poll_ready(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Result<(), Self::Error>>;
    fn start_send(self: Pin<&mut Self>, item: Item)
      -> Result<(), Self::Error>;
    fn poll_flush(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Result<(), Self::Error>>;
    fn poll_close(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Result<(), Self::Error>>;
}
```

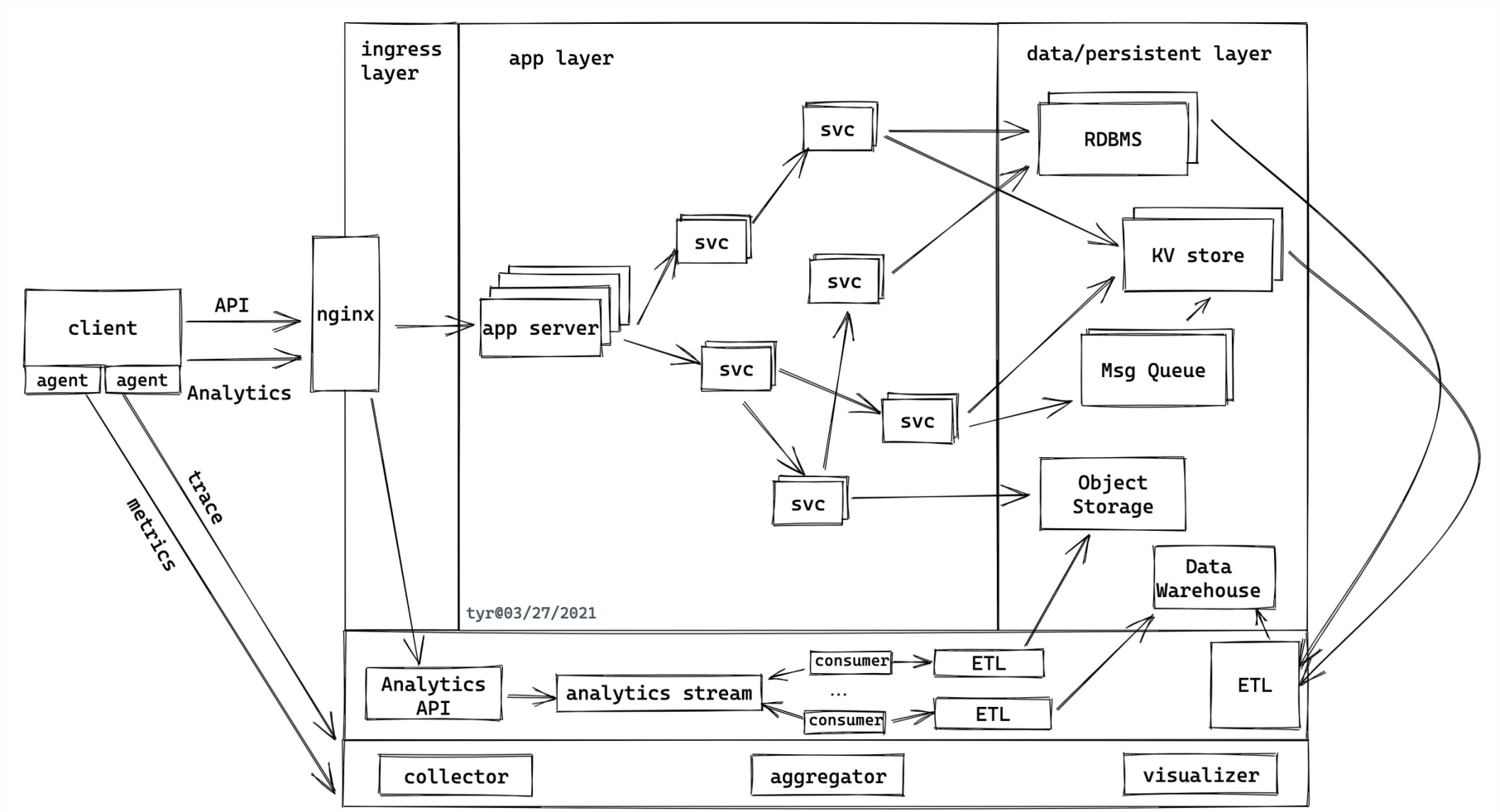
Stream / Sink

网络协议

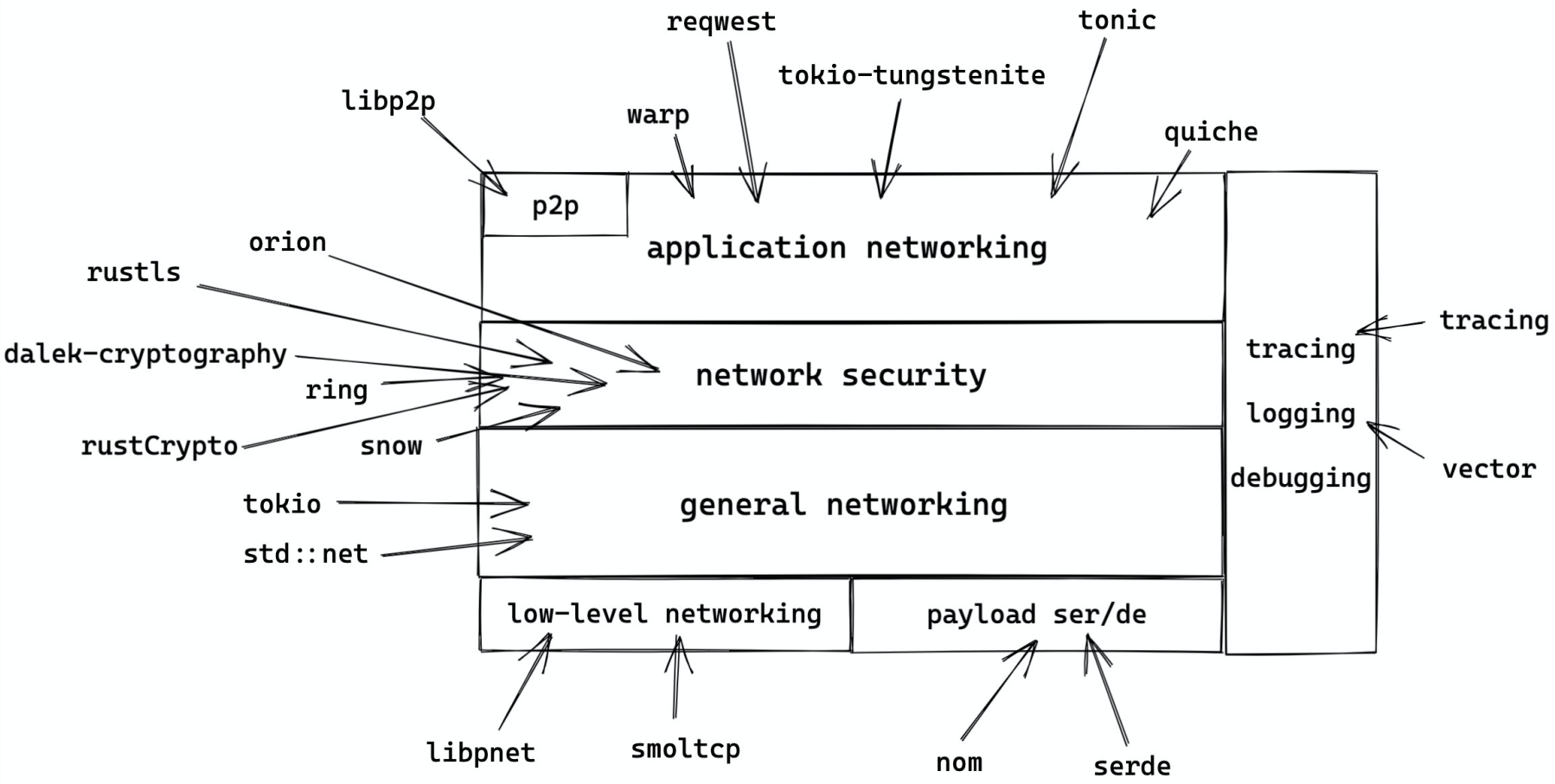
网络协议栈


WebSocket
DNS
MQTT
FTP
...
HTTP
HTTP/2
mDNS
DHCP
DHCPv6
SNMP
L7: Application
TLS
Noise Protocol
...
Socket (@tyr 2021/03/18)
L5: Session
TCP
UDP
RAW
L4: Transport
IP
IPv6
ARP
Auto-IP
NDP
...
ICMP
IGMP
ICMPv6
MLD
L3: Network
ethernet
Wi-Fi
PPP
...
L2: Link

中心化服务的一般设计


ingress layer
app layer
data/persistent layer
client
agent
agent
API
Analytics
nginx
app server
svc
svc
svc
svc
svc
svc
RDBMS
KV store
Msg Queue
Object Storage
Data Warehouse
trace
metrics
tyr@03/27/2021
Analytics API
analytics stream
consumer
...
consumer
ETL
ETL
ETL
collector
aggregator
visualizer

Rust 对网络协议的支持



std::net

tokio:net

Live coding: kv store (基于 tokio)

- 使用 protobuf (prost) 设计 kv store 的传输协议
- 用 tokio TcpListener / TcpStream 实现客户端和服务器的交互
- 使用 dashmap 在内存中存储 kv

家庭作业：用 sled 替换 dashmap

让 kv store 可持久化

服务端应用的基本组成部分

- 数据序列化: serde / protobuf / flatbuffer / capnp / etc.
- 传输协议: tcp / http / websocket / quic / etc.
- 安全协议: TLS / noise protocol / secio / etc.
- 应用协议: your own application logic
- 数据在各个部分之间的流传：共享内存，channel 等

tonic：Rust 下 GRPC 服务

- tower-service
- 基于 prost，生成 ProstCodec
- 为你的 grpc service 定义生成 `Service` trait

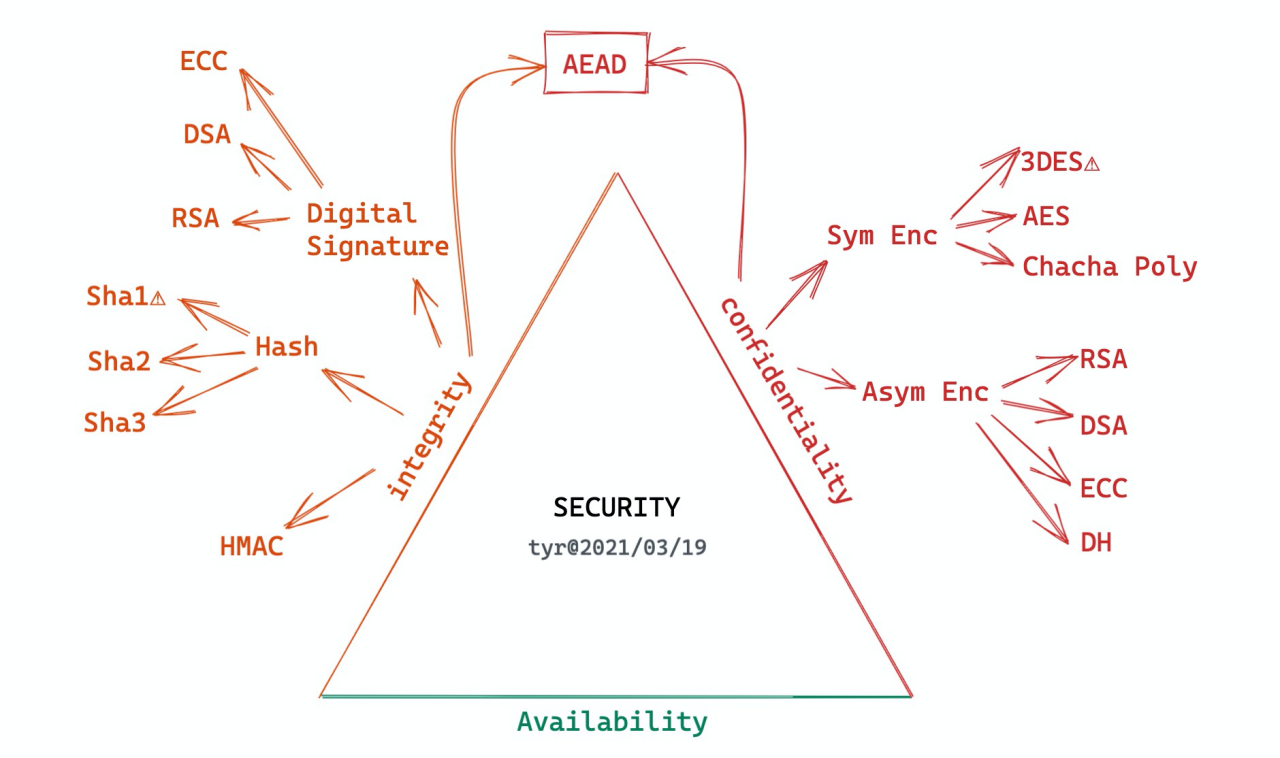
```rust
pub trait Service<Request> {
    type Response;
    type Error;

    type Future: Future<Output = Result<Self::Response, Self::Error>>;
    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>;
    fn call(&mut self, req: Request) -> Self::Future;
}
```

live coding: PoW 服务

- 用 tonic 实现网络层（submit/subscribe）
- 用 std::thread 做计算密集型任务（PoW）
- 用 channel 连接异步/同步世界
- API: `subscribe` / `submit`

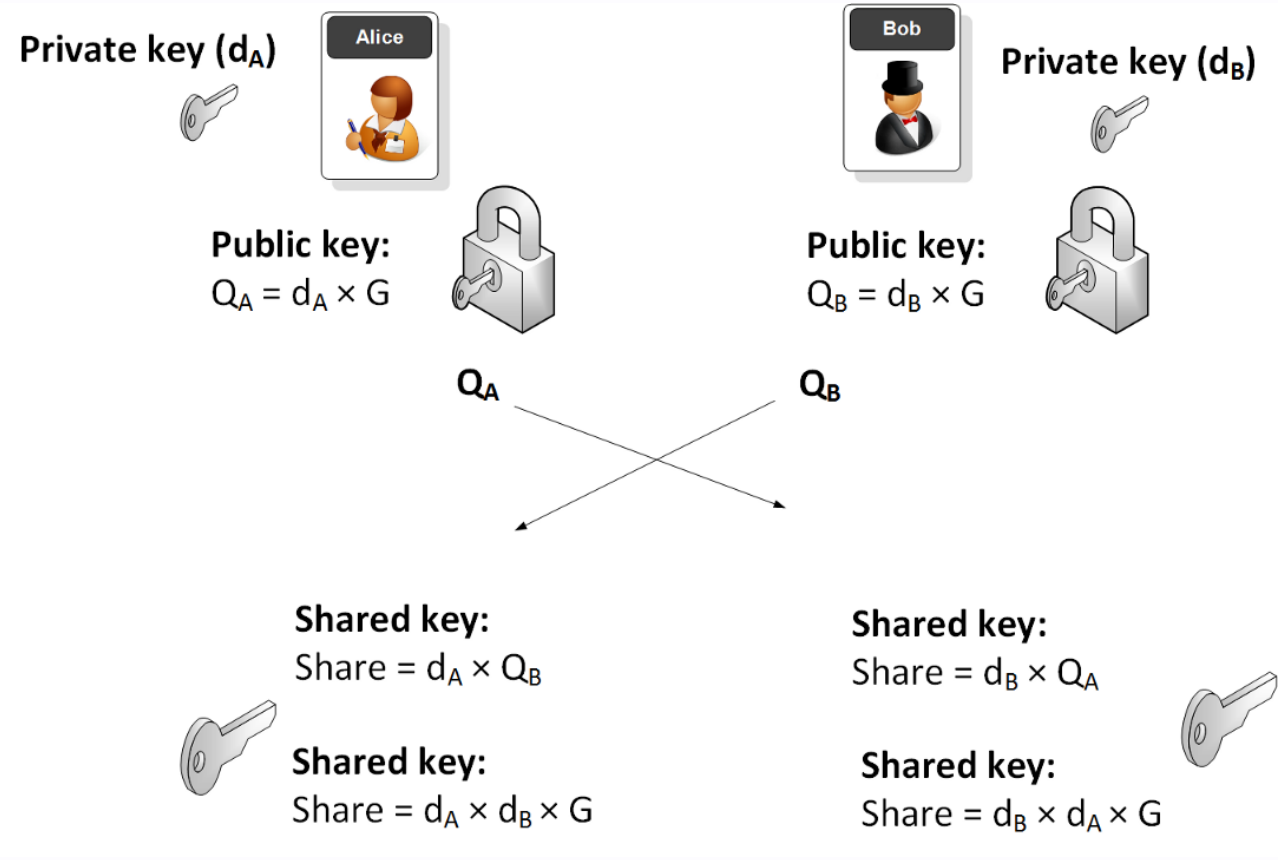
网络安全

当我们谈论网络安全的时候，我们在谈论什么？



应用层安全

- 使用标准协议 - TLSv1.3
- 构建你自己的安全协议 - Noise Protocol
- 应用层安全的基石：DH 算法



Rust TLS 支持

- openssl
- rustls (基于 ring)
- tokio-tls-helper

配置

- 客户端：domain，CA cert
- 服务器：cert / key

```toml
# client configuration

domain = "localhost"

[cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBeTCCASugAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMzEwMzEyMTg0NTU2WjA3MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xEjAQBgNVBAMMCURvbWFpbiBDQTAqMAUGAytlcAMhAAzhorM9IPsXjBTx
ZxykGl5xZrsj3X2XqKjaAVutnf7po1wwWjAUBgNVHREEDTALgglsb2NhbGhvc3Qw
HQYDVR0OBBYEFD+NqChBZDOs5FMgefHJSIWiRTHXMBIGA1UdEwEB/wQIMAYBAf8C
ARAwDwYDVR0PAQH/BAUDAwcGADAFBgMrZXADQQA9sligQcYGaBqTxR1+JadSelMK
Wp35+yhVVuu4PTL18kWdU819w3cVlRe/GHt+jjlbk1i22TvfO5AaNmdxySkO
-----END CERTIFICATE-----"""

# server configuration

[identity]
key = """-----BEGIN PRIVATE KEY-----
MFMCAQEwBQYDK2VwBCIEII0kozd0PJsbNfNUS/oqI/Q/enDiLwmdw+JUnTLpR9xs
oSMDIQAtkhJiFdF9SYBIMcLikWPRIgca/Rz9ngIgd6HuG6HI3g==
-----END PRIVATE KEY-----"""

[identity.cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBazCCAR2gAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMjIwMzE0MTg0NTU2WjA5MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xFDASBgNVBAMMC0dSUEMgU2VydmVyMCowBQYDK2VwAyEALZISYhXRfUmA
SDHC4pFj0SIHGv0c/Z4CIHeh7huhyN6jTDBKMBQGA1UdEQQNMAuCCWxvY2FsaG9z
dDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMEBTADAQEAMA8GA1UdDwEB/wQF
AwMH4AAwBQYDK2VwA0EAy7EOIZp73XtcqaSopqDGWU7Umi4DVvIgjmY6qbJZP0sj
ExGdaVq/7MOlZl1I+vY7G0NSZWZIUilX0CoOkrn0DA==
-----END CERTIFICATE-----"""
```

代码

- 服务器
  - 加载配置 `ServerTlsConfig`
  - 准备好 `TLS acceptor`
  - `acceptor.accept(tcp_stream)`
- 客户端
  - 加载配置 `ClientTlsConfig`
  - 准备好 `TLS connector`
  - `connector.connect(tcp_stream)`

Server:

```rust
// you could also build your config with cert and identity separately. See tests.
let config: ServerTlsConfig = toml::from_str(config_file).unwrap();
let acceptor = config.tls_acceptor().unwrap();
let listener = TcpListener::bind(addr).await.unwrap();
tokio::spawn(async move {
    loop {
        let (stream, peer_addr) = listener.accept().await.unwrap();
        let stream = acceptor.accept(stream).await.unwrap();
        info!("server: Accepted client conn with TLS");

        let fut = async move {
            let (mut reader, mut writer) = split(stream);
            let n = copy(&mut reader, &mut writer).await?;
            writer.flush().await?;
            debug!("Echo: {} - {}", peer_addr, n);
        }

        tokio::spawn(async move {
            if let Err(err) = fut.await {
                error!("{:?}", err);
            }
        });
    }
});
```

Client:

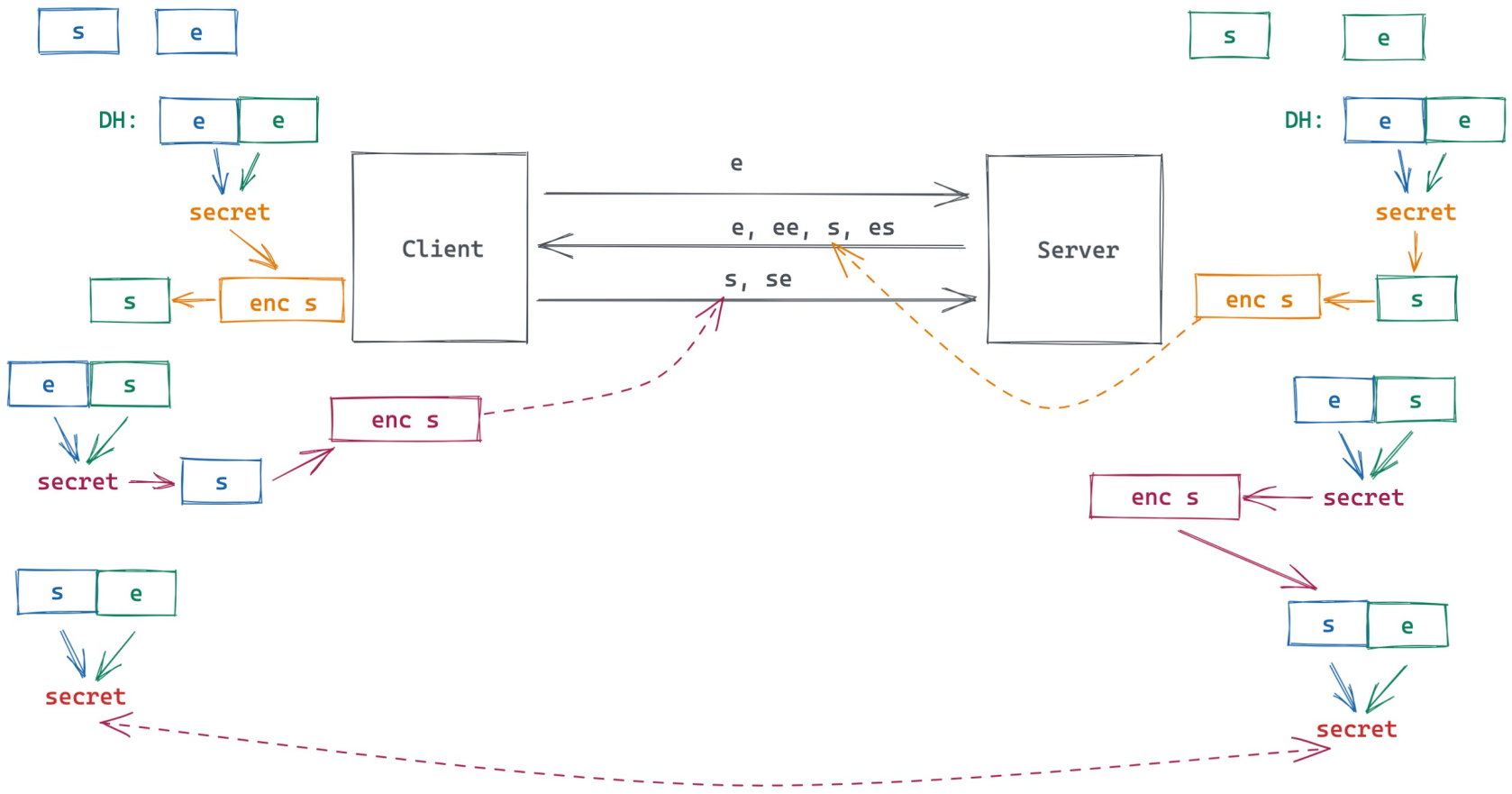
```rust
let msg = b"Hello world\n";
let mut buf = [0; 12];

// you could also build your config with cert and identity separately. See tests.
let config: ClientTlsConfig = toml::from_str(config_file).unwrap();
let connector = config.tls_connector(Uri::from_static("localhost")).unwrap();

let stream = TcpStream::connect(addr).await.unwrap();
let mut stream = connector.connect(stream).await.unwrap();
info!("client: TLS conn established");

stream.write_all(msg).await.unwrap();

info!("client: send data");

let (mut reader, _writer) = split(stream);

reader.read_exact(buf).await.unwrap();

info!("client: read echoed data");
```

Noise Protocol

- TLS vs Noise protocol：动态协商 vs 静态协商
- Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s：
  - I：发起者的固定公钥未加密就直接发给应答者
  - K：应答者的公钥发起者预先就知道
  - psk2：把预设的密码（Pre-Shared-Key ）放在第 2 个握手包之后
  - ChaChaPoly：对称加密算法
  - BLAKE2s：哈希算法
- 协议最少 0-RTT (x 或者 xpsk），之后就建立好加密通道，可以发送数据

Noise Protocol 接口

- build：根据协议变量和固定私钥，初始化 HandshakeState
- write(msg, buf): 根据当前的状态，撰写协议报文或者把用户传入的 buffer 加密
- read(buf, msg)：根据当前的状态，读取用户传入的 buffer，处理握手状态机或者把用户传入的 buffer 解密
- into_transport_mode：将 `HandshakeState` 转为 `TransportState`
- rekey：在传输模式下，用户可以调用 rekey 来更新密钥


s
e
DH:
e
e
secret
s
enc s
Client
e
e, ee, s, es
s, se
Server
s
e
DH:
e
e
secret
enc s
s
e
s
enc s
secret
s
e
s
secret
enc s
secret
s
e
secret
s
e
secret

代码（0-RTT）

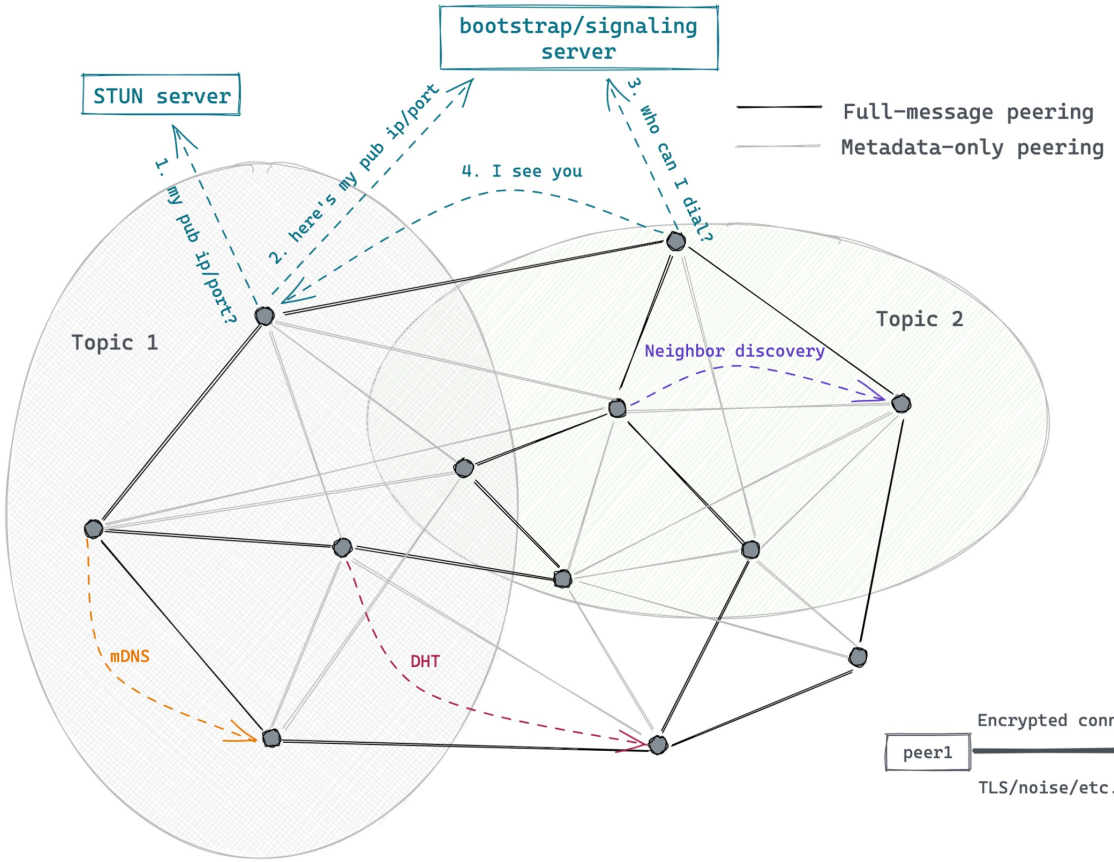
- Initiator:
  - 构建 `HandshakeState`
  - 发送握手数据
  - 进入传输模式
- Responder:
  - 构建 `HandshakeState`
  - 接收握手数据
  - 进入传输模式

```rust
pub fn new(config: SessionConfig) -> Result<Self, ConcealError> {
    let mut header: Header = config.header;
    let noise_params: NoiseParams = header.to_string().parse()?;
    // in handshake mode this should be enough
    let mut buf: [u8; _] = [0u8; 256];

    if header.handshake_message.is_empty() {
        // initiator
        let mut noise: HandshakeState = if !header.use_psk {
            Builder::new(noise_params): Builder
                .remote_public_key(pub_key: &config.rs.unwrap()): Builder
                .local_private_key(&config.keypair.private): Builder
                .build_initiator()?
        } else {
            Builder::new(noise_params): Builder
                .remote_public_key(pub_key: &config.rs.unwrap()): Builder
                .local_private_key(&config.keypair.private): Builder
                .psk(location: 1, key: &config.psk.unwrap()): Builder
                .build_initiator()?
        };

        let len: usize = noise.write_message(payload: &[0u8; 0], message: buf.as_mut())?;
        let handshake_message: Vec<u8> = buf[..len].to_vec();
        header.handshake_message = handshake_message;
        let state: TransportState = noise.into_transport_mode()?;
        Ok(Self { state, header })
    } else {
        // responder
        let mut noise: HandshakeState = if !header.use_psk {
            Builder::new(noise_params): Builder
                .local_private_key(&config.keypair.private): Builder
                .build_responder()?
        } else {
            Builder::new(noise_params): Builder
                .local_private_key(&config.keypair.private): Builder
                .psk(location: 1, key: &config.psk.unwrap()): Builder
                .build_responder()?
        };
        let _len: usize = noise.read_message(&header.handshake_message, payload: &mut buf)?;
        let state: TransportState = noise.into_transport_mode()?;
        Ok(Self { state, header })
    }
}
```

live coding: 使用 noise protocol 增强 kv store 的安全性

- 实现 NoiseCodec?
- 实现 Stream / Sink?

P2P 应用的一般结构



Network identity: normally a keypair

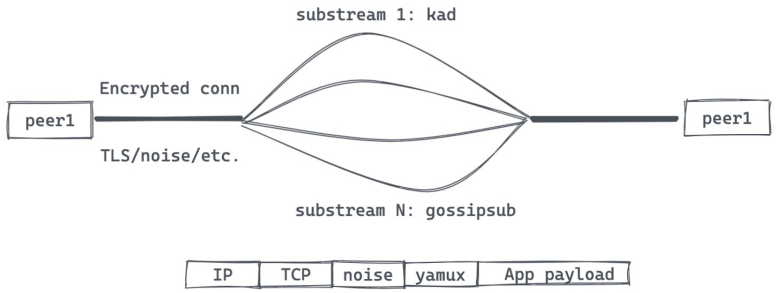
Transport: TCP/QUIC/WS/WebRTC ...

Security: TLS / noise

Multiplex: Yamux/Mplex/QUIC

Protocol:

- peer discovery: mDNS, bootstrap, KadDHT
- peer routing: Kad DHT
- content discovery: pubsub, Kad DHT
- NAT traversal: STUN/TURN/ICE
- App protocol: think gRPC



Live coding：p2p 聊天

- 使用 mDNS 做本地节点的发现
- 使用 gossipsub 广播数据

参考资料

- GRPC Protocol
- Are we web yet?
- Tonic: rust grpc framework
- snow
- Rust 的 Pin 与 Unpin

宏编程

声明宏

- `macro_rules!`
- `#[macro_use]` / `#[macro_export]`

Live coding: `vec!` 和 `prost_into_vec!`

过程宏

- 类函数宏（function-like macros）： `println!(...)`
- 派生宏（derive macros）： `#[derive(Debug)]`
- 标记宏（attribute macros）： `#[tokio::main]`

Live coding: 使用 derive macro 实现 builder pattern

参考资料

- mscros by example
- syn
- quote

FFI with C/Elixir/Swift/Java

WASM/WASI

Rust for real-world problems

May the Rust be with you