程序设计训练（Rust 语言）

第 5 讲：工程管理与常用库

韩文弢

清华大学

2024–2025 学年度夏季学期

本讲内容

5.1 工程管理

5.1.1 工程管理的需求

- 随着工程（project）变大，参与者变多，组织代码的方式会越来越重要。
- 封装可以隐藏实现细节，使用户更好地复用代码。
- 项目（item）的作用域管理是需要支持的核心功能。
  - Rust 中，项目指的是模块、函数、类型、常量等。

5.1.2 模块

- Rust 代码的组织结构基于模块（module），模块组成树状结构。
- 一般来说，每个文件是一个模块，例如模块 `foo` 对应于文件 `foo.rs`。
- 模块的树状结构通过 `mod` 语句连接，例如 `mod foo` 表示模块 `foo` 是当前模块的一个子模块。
- 除了将模块写在一个单独的文件，也可以将模块写在一个代码块：

```rust
mod foo {
    // 模块 foo 的内容，等效于将这里的代码写在 foo.rs 中
}
```

5.1.3 模块（续）

- 顶层模块一般位于 `src/main.rs` 或 `src/lib.rs`，根模块的子模块 `foo` 位于 `src/foo.rs`，`foo` 模块的子模块 `bar` 则位于 `src/foo/bar.rs`。
- `foo.rs` 也可以放在 `src/foo/mod.rs`，但这是不推荐的做法（同名文件更难区分）。

5.1.4 可访问性管理

- Rust 中各个类型、函数、结构体字段以及子模块的可访问性都基于模块。
- 一个模块中定义的各种项目默认只有当前模块以及子模块可以访问。
- 可以通过在项目和结构体的字段前标注 pub 使得项目或结构体的字段在模块外也可访问。

5.1.5 项目访问路径

- 直接访问项目时，默认使用相对路径，起始于当前模块：

```rust
mod one {
    mod two { pub fn foo() {} }
    fn bar() {
        two::foo();
    }
}
```

- self / super / crate 分别表示当前模块、上层模块、顶层模块：

```rust
self::two::foo
super::one::two::foo
crate::one::two::foo
```

5.1.6 通过 use 引入项目

- 可以使用 use 将项目引入到当前模块的作用域中，从而可以直接访问。

```rust
mod one {
    mod two {
        pub fn foo() {}
    }
    use two::foo;  // 将 two::foo 引入当前模块
    fn bar() {
        foo();  // 可以直接使用 foo
    }
}
```

5.1.7 通过 use 引入项目（续）

- 可以在一条 use 语句中通过大括号同时引入多个项目，或者通过 * 引入所有项目（谨慎使用）：

```rust
use one::{two::foo, bar};
use one::*;
```

5.1.8 重新导出

可以使用 pub use 来重新导出其他模块中的项目，等效于该项目就定义在本模块中：

```rust
mod one {
    mod two {
        pub fn foo() {}
    }
    pub use two::foo;  // 重新导出
}
fn baz() {
    one::foo();
}
```

5.1.9 箱（crate）

- 每棵模块树构成一个**箱**（crate）。
- **包**（package）是由 `Cargo.toml` 文件定义的一套 Rust 代码，一个包可以有零或一个**库箱**（library crate）和任意个**二进制箱**（binary crate）。
- 库箱可以在其他包中作为依赖被调用（即访问顶层模块中暴露的项目）。库箱的顶层模块默认位于 `src/lib.rs`。
- 二进制箱的顶层模块中有 `main` 函数，可以运行。二进制箱的顶层模块默认位于 `src/main.rs`。

5.1.10 代码组织

即使不打算在其他包中引用库箱，也可以

- 将核心逻辑写在库箱中
- 在二进制箱中引用库箱

这样能

- 让代码组织结构更加清晰
- 可以在多个二进制箱中共用代码
- 方便进行集成测试

5.1.11 使用库箱

- 运行命令 `cargo add rand` 或者手动编辑 `Cargo.toml` 来使用 crates.io 上名为 `rand` 的库箱：

```toml
[dependencies]
rand = "0.9.2"
```

- 通过库箱的名称访问其内容，如：`use rand::rng`。
- 在同一个包中，二进制箱里访问库箱也是通过库箱的名称（`Cargo.toml` 中的 package name）来访问。

5.1.12 使用自己写的箱

除了将库箱发布到 crates.io，也可以使用 Git 仓库或本地文件系统中的库箱：

```toml
[dependencies]
myfoo = { git = "https://github.com/me/myfoo" }
mybar = { path = "../mybar" }
```

5.1.13 Cargo：默认代码位置

- 多个二进制箱可以放在 `src/bin/*.rs`，作为可执行文件。
  - 可以用 `cargo run --bin foo` 来指定运行的二进制箱。
- 示例代码放在 `examples/*.rs`。
  - 会在 `cargo test` 的时候构建，保证示例代码可以通过编译。
  - 可以用 `cargo run --example foo` 来运行。
- 集成测试（非单元测试）放在 `tests/*.rs`，用来测试正确性。
- 基准测试程序 (benchmarks) 放在 `benches/*.rs`，用来测试性能。

5.1.14 Cargo：特性

- 箱的特性（feature）是可以在构建时做选择性的开关。
- 使用箱作为依赖时，指定启用哪些特性：

```
cargo add rand --features=serde
```

- 在编写箱时支持特性：
  - 在 `Cargo.toml` 中的 `[features]` 里列出特性。
  - 可以将部分依赖设为可选依赖，仅在启用特性时使用可选依赖。
  - 二进制箱可以依赖于特性，只有启用了一些特性时才能构建某个二进制箱。

5.1.15 Cargo：构建脚本

- 如果遇到超过 Cargo 所提供的功能的特殊需求，可以通过构建脚本（build scripts）来实现。构建脚本也是用 Rust 语言编写的。

```toml
[package]
build = "build.rs"
```

- 当构建脚本存在时，`cargo build` 会先编译并运行构建脚本。

5.1.16 Cargo：更多功能

- `cargo clippy`：运行 clippy 静态分析工具。
- `cargo fmt`：格式化代码。
- `cargo doc`：生成文档。
- `cargo install`：从 crates.io 上安装二进制文件。
- 把自己写的软件包发布到 crates.io 上。
- 创建工作空间（workspaces），组织多个包协同开发。
- 通过 `cargo-<command>` 来扩展功能。
- Cargo 提供了很多有用的功能，详见 The Cargo Book。

5.2 语法补充

5.2.1 属性

- 属性 (attributes) 是 Rust 代码给编译器传递信息的机制。
  - 例如，`#[test]` 属性用于将函数标注为测试用例。
- `#[foo]` 标注随后的代码段，`#![foo]` 标注所在的代码段。

```rust
#[foo]
fn midterm1() {
    // ...
}
fn midterm2() {
    #![foo]
    // ...
}
```

5.2.2 常见属性

- `#![no_std]` 禁用标准库（用于嵌入式开发等没有标准库的环境）。
- `#[derive(Debug)]` 运行派生宏（例如自动获得特型）。
- `#[inline(always)]` 提示编译器的内联优化行为。
- `#[allow(missing_docs)]` 屏蔽编译器的某些警告。
- `#![crate_type = "lib"]` 提供箱的元数据。
- `#![feature(box_syntax)]` 启用不稳定版本（nightly）的语法。
- `#[cfg(feature = "foo")]` 定义条件编译（启用箱的特性时编译）。
- `#[cfg(target_os = "linux")]` 定义条件编译（在特定操作系统上编译）。

更多属性可见参考手册。

5.2.3 操作符

Rust 语言中操作符的优先级顺序如下：

- 函数调用、下标索引：`()[]`
- 错误传播：`?`
- 单目操作符：`! - * & &mut`
- 类型转换：`as`
- 乘法类：`* / %`
- 加法类：`+ -`
- 位移类：`<< >>`
- 按位与：`&`
- 按位异或：`^`
- 按位或：`|`
- 比较类：`== != < > <= >=`
- 逻辑与：`&&`
- 逻辑或：`||`
- 范围：`.. ..=`
- 赋值：`= += -=` 等等

5.2.4 操作符重载

Rust 使用特型来重载操作符，定义在 `std::ops` 下。

- `Neg`, `Not`, `Deref`, `DerefMut`
- `Mul`, `Div`, `Mod`
- `Add`, `Sub`
- `Shl`, `Shr`
- `BitAnd`, `BitXor`, `BitOr`
- `Eq`, `PartialEq`, `Ord`, `PartialOrd`
- `And`, `Or`

此外还有：`Fn`, `FnMut`, `FnOnce`, `Index`, `IndexMut`

5.2.5 类型转换

- as 操作符不能重载。
- 使用 From 和 Into 实现自定义类型转换。
  - `trait From<T> { fn from(T) -> Self; }`，调用形式为 `Y::from(x)`。
  - `trait Into<T> { fn into(self) -> T; }`，调用形式为 `x.into()`。
- 如果实现了 From，则 Into 会自动实现。建议优先实现 From。
  - 也就是说，`From<T> for U` 蕴含 `Into<U> for T`。
- 此外，还有 TryFrom 和 TryInto，用于可能失败的转换。

5.2.6 自定义类型转换示例

Rust

```rust
struct A(Vec<i32>);
impl From<Vec<i32>> for A {
    fn from(v: Vec<i32>) -> Self {
        A(v)
    }
}
```

5.2.7 命名规范：标识符

- `CamelCase`：类型、特型
- `snake_case`：箱、模块、函数、方法、变量
- `SCREAMING_SNAKE_CASE`：常量和静态变量
- `T`（单个大写字母）：类型参数
- `'a`（撇 + 短的小写名字）：生命周期参数

5.2.8 命名规范：构造函数和转换函数

- `new`, `new_with_stuff`：构造函数
- `from_foo`：转换构造函数，例如 `from_str`
- `as_foo`：低开销非消耗性转换，例如 `as_ref`
- `to_foo`：高开销非消耗性转换，例如 `to_string`
- `into_foo`：消耗性转换，例如 `into_iter`

5.3 智能指针

5.3.1 &T 和 &mut T

- 基本的、经济型的引用
- 无运行时开销，所有检查在编译时完成。
- 具有固定的生命周期，没有灵活性。
- 如果可以，尽量使用引用。

- Box<T> 用于在堆上分配空间存放数据。
- Box<T> 拥有 T 类型的对象，它的指针是唯一的（类似 C++ 的 std::unique_ptr），不能创建别名（可以借用）。
- 当 Box<T> 超过作用域时，对象自动释放。
- 使用方法和不加 Box<T> 的一般类型相似，主要区别是动态分配。
- 通过 Box::new() 来创建。

```rust
let boxed_five = Box::new(5);
```

5.3.3 Box<T> 的特点

- 优点：
  - 是使用堆的最简单的办法。
  - 是动态分配的零开销抽象。
  - 借用和移动语义同样适用。
  - 自动销毁。
- 缺点：
  - `Box` 严格拥有里面的 `T` 类型的对象，是唯一的所有者。
  - 如果有引用的话，`Box` 本身必须存活到所有引用失效之后。

5.3.4 Box::leak()

`Box::leak()` 可以将 `Box<T>` 转换为 `&'static mut T` 类型，使得数据可以在程序余下的整个生命周期中使用。

```rust
let boxed_five = Box::new(5);
let static_five: &'static _ = Box::leak(boxed_five);
```

可用于全局变量的初始化，例如程序的配置信息。

5.3.5 std::rc::Rc<T>

- `Rc<T>` 是共享所有权的指针类型，相当于 C++ 中的 `std::shared_ptr`。
- 是“Reference Counted”的缩写，记录指针的别名个数。
- 调用 `Rc` 的 `clone()` 方法来获得新的引用。
  - 会增加引用计数。
  - 不会拷贝数据。
- 当引用计数降为 0 时，会释放对象。
- `T` 的值只有在引用计数为 1 时才能修改，与 Rust 的借用规则一致。

5.3.6 Rc 使用示例

Rust

```rust
let mut shared = Rc::new(6);
// 只有一个 Rc 时允许修改
println!("{:?}", Rc::get_mut(&mut shared)); // ==> Some(6)
// 创建该值的另一个引用
let mut cloned = shared.clone();
// 有多个 Rc 时不允许修改
println!("{:?}", Rc::get_mut(&mut shared)); // ==> None
println!("{:?}", Rc::get_mut(&mut cloned)); // ==> None
```

- 当有环时，引用计数会发生问题，例如：
  - A 有一个 B 的 Rc，B 也有一个 A 的 Rc，两者的引用计数都是 1。
  - 由于构成了环，两个对象都不会被释放，从而引起内存泄露。
- 可以通过引入**弱引用**来避免。
  - 弱引用不会增加**强引用**的计数。
  - 这也表示弱引用不一定总是有效的。
- Rc 可以通过 Rc::downgrade() 降级成 Weak。
  - 需要访问时，使用 weak.upgrade() -> Option<Rc<T>> 再升级回 Rc。
  - 除此之外，Weak 干不了别的事情，通过升级步骤避免使用无效状态。

5.3.8 强引用和弱引用

- 一般情况下，使用的时候需要通过 `Rc` 获得强引用。
- 如果应用场景中需要访问数据而没有所有权，就需要用到 `Weak`。
  - `Weak` 升级时可能会得到 `None`，需要应对这种情况。
- 引用成环的情况也需要用 `Weak` 来避免内存泄露。
  - 由于可变性规则，`Rc` 的环在 Rust 中很难形成。
- 考虑图的情况，保存顶点和边的时候哪里用 `Rc`，哪里用 `Weak`?

5.3.9 std::rc::Rc<T> 的特点

- 优点：
  - 允许共享数据的所有权。
- 缺点：
  - 有一定的运行时开销。
    - 维护两个引用计数（强和弱）。
    - 需要动态更新和检查引用计数。
  - 引用成环会造成内存泄露。

5.3.10 Cell 和 RefCell

- 允许**内部可变性**的机制。
- 通过不可变引用实现修改所含值的操作。
- 有两种常用的类型：`Cell<T>` 和 `RefCell<T>`。
- 此外，还有 `OnceCell<T>`。

```rust
struct Foo {
    x: Cell<i32>,
    y: RefCell<u32>,
}
```

5.3.11 std::cell::Cell<T>

- 为 Copy 类型提供内部可变性的格子类型。
- 用 get() 从 Cell 中取值。
- 用 set() 更新 Cell 中的值。
  - 不能修改 T 的值，只能整个替换。
- 作用比较受限，但是安全、低开销。

```rust
let c = Cell::new(10);
c.set(20);
println!("{}", c.get()); // 20
```

5.3.12 `std::cell::Cell<T>` 的特点

- 优点：
  - 实现内部可变性。
  - 没有运行时开销。
  - 额外空间开销较小。
- 缺点：
  - 较为受限，只能用于 `Copy` 类型。

5.3.13 std::cell::RefCell<T>

- 可以为任意类型提供内部可变性的格子类型。
- 使用动态借用检查规则，在运行时进行。
  - 有可能会引起运行时的恐慌。
- 通过 `borrow()` 或 `borrow_mut()` 来借用内部数据。
  - 如果 `RefCell` 已经被借用，可能会引起恐慌。

5.3.14 RefCell 使用示例

Rust

```rust
use std::cell::RefCell;

let refc = RefCell::new(vec![12]);
let mut inner = refc.borrow_mut();
inner.push(24);
println!("{:?}", *inner); // [12, 24]

let inner2 = refc.borrow();
// ==> Panics since refc is already borrow_mut'd!
```

5.3.15 std::cell::RefCell<T> 的常见用法

- RefCell 的常见用法是把它放在 Rc 里，这样可以允许共享的可变性。
- 这样做不是线程安全的，borrow() 等方法不能防止数据竞争。
- 可以使用 try_borrow() 等方法来检查借用是否成功。

5.3.16 std::cell::RefCell<T> 的特点

- 优点：
  - 为任意类型提供内部可变性
- 缺点：
  - 要保存额外的借用状态。
  - 要在借用时检查状态。
  - 有可能会引起恐慌。
  - 不是线程安全的。

5.3.17 std::cell::Ref<T> 和 RefMut<T>

- 当 borrow() 一个 RefCell<T> 时，得到的是 Ref<T>，而不是 &T。
  - 同样的，borrow_mut() 返回 RefMut<T>。
- Ref 和 RefMut 包装了普通的引用，提供一些额外的方法。

5.3.18 *const T 和 *mut T

- 像 C 语言一样的裸指针，只是指向内存中的某个地址。
- 没有所有权规则，没有生命周期规则。
- 零抽象代价，因为没有抽象。
- 解引用时需要 unsafe，可能会引起不安全的后果。
- 可以在编写底层代码时使用（例如维护 Vec<T> 的内部状态），一般不用。
  - 使用得当可以避免运行时代价。

5.3.19 自动解引用

Rust 在大多数情况下会对变量做自动解引用操作。

- 在对一个引用进行方法调用的时候。
- 将引用作为函数参数进行传递的时候。

```rust
/// `length` only needs `vector` temporarily, so it is borrowed.
fn length(vec_ref: &&Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
}
fn main() {
    let vector = vec![];
    length(&&&&&&&&&&&&&vector);
}
```

5.3.20 Deref 自动转换

- Rust 的自动解引用行为也可以在类型之间工作。

```rust
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

- 因为 `String` 实现了 `Deref<Target=str>`，所以 `&String` 的值在需要的时候会自动解引用成 `&str`。

5.3.21 制造引用

- `Borrow`/`BorrowMut`：用来借用数据的特型。

```rust
trait Borrow<Borrowed> {
    fn borrow(&self) -> &Borrowed;
}
```

- `AsRef`/`AsMut`：轻量级的引用到引用的转换。

```rust
trait AsRef<T> {
    fn as_ref(&self) -> &T;
}
```

- 它们的作用一样吗？

5.3.22 不同制造引用方法的区别

- `Borrow` 会有更多的附加含义：
  - 如果实现 `Borrow` 且 `Self` 和 `Borrowed` 都实现了 `Hash`、`Eq`/`Ord`，则这些功能的结果必须是一致的。
- `Borrow` 有兜底实现 `impl<T> Borrow<T> for T`，使得从可以从 `T` 转换成 `&T`。
- `AsRef` 也有兜底实现 `impl<'a, T, U> AsRef<U> for &'a T where T: AsRef<U>`。
  - 对于所有 `T`，如果 `T` 实现了 `AsRef`，那么 `&T` 也实现了 `AsRef`。
- 如果要制造引用，通常应该实现 `AsRef`。

5.4 常用库

5.4.1 常用第三方库

- 正则表达式：regex
- 日志：log
- 日期：chrono
- HTTP 客户端：reqwest、ureq
- 增强错误处理：thiserror、anyhow、color-eyre
- 数据库：rusqlite、r2d2、sqlx、diesel、sea-orm

更多优秀的第三方库可参见 blessed.rs、Awesome Rust。

5.4.2 错误处理的烦恼

- 考虑如下场景：想要开发一个库发布给别人使用。
- 代码里可能会出现一些需要报告错误的场景，此时通常需要自定义一个错误类型：

```rust
enum MyError {
    MyCustomError,
    MyCustomError2(String),
}
impl std::error::Error for MyError { }
```

- 但是实现起来比较烦琐：需要根据错误类型，进行模式匹配，然后格式化为适合用户阅读的错误信息。

5.4.3 thiserror

利用 thiserror 可以方便地自定义一个新的错误类型：

Rust

```rust
use thiserror::Error;
#[derive(Error, Debug)]
pub enum DataStoreError {
    #[error("data store disconnected")]
    Disconnect(#[from] io::Error),
    #[error("the data for key `{0}` is not available")]
    Redaction(String),
    #[error("invalid header (expected {expected:?}, found {found:?})")]
    InvalidHeader {
        expected: String,
        found: String,
    },
}
```

5.4.4 anyhow

如果你编写的是应用程序，可能需要处理来自不同库的错误，这些错误的类型不同，可以用 anyhow 简化代码：

```rust
use anyhow::Result;

fn get_cluster_info() -> Result<ClusterMap> {
    let config = std::fs::read_to_string("cluster.json")?;
    let map: ClusterMap = serde_json::from_str(&config)?;
    Ok(map)
}
```

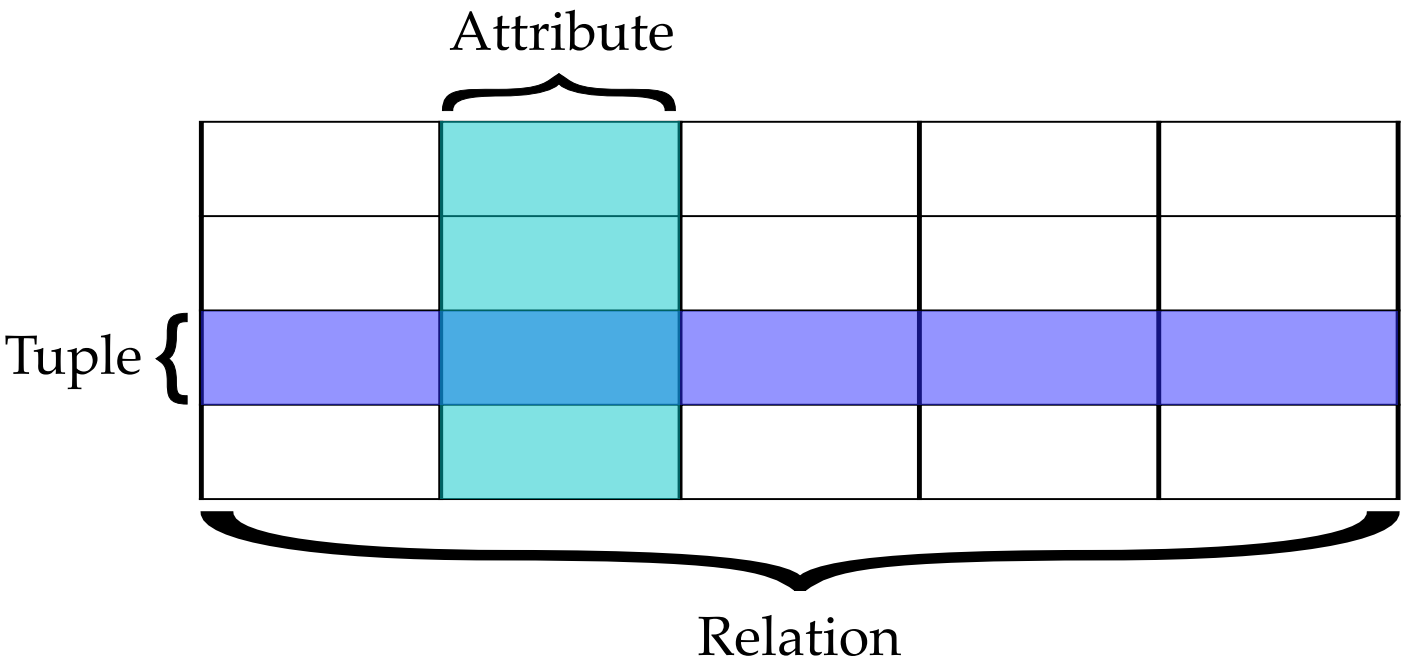
能够自动把不同类型的错误转换为 anyhow::Error

5.4.5 数据库

- 数据库 (database) 是以一定方式储存在一起、能予多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。
- 用户可以对集合中的数据执行新增、截取、更新、删除等操作。
- 数据库的分类：
  - 关系数据库 (relational database)：
    - PosgreSQL、MySQL、SQLite、……
  - 非关系型数据库 (NoSQL)：
    - 文档数据库 (document DB)，如 MongoDB。
    - 键值数据库 (key-value DB)，如 LevelDB。
    - ……

5.4.6 关系数据库

关系数据库是创建在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。



5.4.7 关系数据库的操作

- 数据查询
  - 选择
  - 投影
  - 连接
  - 并、交、差……
- 数据操作
  - 新增
  - 删除
  - 修改
  - 查询
- 数据库查询语言 SQL (Structured Query Language，读作 sequel)

5.4.8 SQL 简介

- 常用命令
  - 创建表格（关系）：`CREATE TABLE`
  - 查询数据：`SELECT`
  - 插入数据：`INSERT`
  - 更新数据：`UPDATE`
  - 删除数据：`DELETE`
  - 删除表格：`DROP TABLE`

5.4.9 创建表格

```sql
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
CREATE TABLE PhoneNumbers (
    RecordID int,
    PhoneNumber varchar(11),
    PersonID int
);
```

5.4.10 查询数据

```sql
SELECT * FROM Persons;
```

```sql
SELECT LastName, FirstName, Address FROM Persons
    WHERE City = 'Beijing';
```

```sql
SELECT LastName, FirstName, PhoneNumber
    FROM Persons, PhoneNumbers
    WHERE Persons.PersonID = PhoneNumbers.PersonID
    ORDER BY PhoneNumber;
```

5.4.11 插入数据

```sql
INSERT INTO Persons VALUES (
    1,
    'Han',
    'Wentao',
    'Tsinghua University',
    'Beijing'
);
```

5.4.12 更新数据

```sql
UPDATE Persons SET Address = 'Unknown'
    WHERE City = 'Utopia';
```

5.4.13 删除数据

```sql
DELETE FROM Persons WHERE Address = '';
```

SQL

当心下面的 SQL 语句！

```sql
DELETE FROM Persons;
```

SQL

5.4.14 删除表格

```sql
DROP TABLE Persons;
```

5.4.15 在 Rust 中使用 SQL

软件包 rusqlite：

```rust
fn main() -> Result<()> {
    let conn = Connection::open_in_memory()?;

    conn.execute(
        "CREATE TABLE person (
            id   INTEGER PRIMARY KEY,
            name TEXT NOT NULL,
            data BLOB
        )",
        (), // empty list of parameters.
    )?;
}
```

5.4.16 与 Web 框架联合使用

- 创建数据库连接池 (r2d2, r2d2-sqlite)。
- 将连接池作为 `Data<T>` 传给请求处理代码。
- 详见 Actix 文档。

5.5 小结

5.5.1 本讲小结

- 模块、箱和包
- Cargo 高级用法
- 智能指针
- 常用第三方库