Rust课程III
闭包 / 迭代器 / 智能指针 / 并发

徐启航 西安交通大学

闭包

Closures

闭包

- 本质：拥有可能的关联上下文的匿名函数结构体。

- 示例：定义和使用闭包
- 示例：展开闭包的实际实现

- move关键字：强制捕获上下文所有权
- 示例：使用move关键字

闭包的类型推断

```
fn main() {
    let a = 15;
    let c1 = |x: i32| -> i32 { a + x };
    let c2 = |x: i32|        { a + x };
    let c3 = |x|             { a + x };
    let c4 = |x|               a + x  ;
    assert_eq!(c1(0), c2(0));
    assert_eq!(c3(0), c2(0));
    assert_eq!(c3(0), c4(0));
}
```

闭包的特征表示与存储

- FnOnce – 调用该对象消耗其上下文的所有权
- FnMut – 调用该对象需要其上下文的可变引用
- Fn – 调用该对象仅需要其上下文的不可变引用
- 所有的函数（fn）实现了Fn特征，相当于没有捕获上下文的闭包。

- 示例：将闭包作为函数参数或者返回值
- 示例：将闭包存储到数据结构中

迭代器

Iterators

迭代器

- 本质：对序列的流式处理——变换或消耗。
- 任务：
  - 遍历序列的每个项
  - 控制遍历的时机
- Rust的迭代器是懒惰的——不消耗迭代器就不会遍历。

- 示例：消耗迭代器进行遍历的基本方法

迭代器的特征

```rust
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // Other provided methods
}
```

- 示例：使用特征方法消耗迭代器
- 示例：自定义计数器迭代器
- 示例：展开for循环

迭代器的变换和消耗（常用示例）

- 迭代器变换：将迭代器变换为新的迭代器（链式调用）
  - map –将每一个项映射成新的数据
  - filter – 过滤不满足条件的项
  - zip、chain – 将两个迭代器并联、串联
  - skip、take – 跳过特定数量的项、仅取特定数量的项
- 迭代器消耗：消耗迭代器，产生对项组合的结果
  - sum、product、max、min – 对所有项进行求和、求积、求最值
  - for_each – 遍历所有项，for循环的平替
  - reduce – 对项进行两两组合得到可能的唯一结果
  - collect – 将所有项收集到某个新集合中
- 示例：使用迭代器的变换和消耗

迭代器的性能

- 与循环相比：稍快，数量级相同
- 零开销抽象

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

智能指针

Smart pointers

智能指针

- 引用的本质：指针
- 智能指针的本质：可能具有额外功能的模拟引用（Deref）。可能拥有所有权。

- 示例：使用Box、Vec、String等基本智能指针

智能指针的特征

```rust
pub trait Deref {
    type Target: ?Sized;

    // Required method
    fn deref(&self) -> &Self::Target;
}
```

- 示例：查看Box、Vec、String的对应实现
- 示例：自定义简单的智能指针
- 示例：可变引用的DerefMut

智能指针：引用计数

- Rc（Reference Counting）的本质：共享所有权
- 由于借用规则，没有DerefMut实现，无法获得可变引用
- 使用不当会造成循环引用，导致内存泄漏
  - 使用Weak解决

- 示例：使用Rc创建共享数据的单向链表
- 示例：使用Rc和Weak创建双向链表

内部可变性

- 本质：在持有不可变引用的情况下对内部数据进行修改
- 将unsafe的代码实现包装成了safe API
- 与借用规则冲突？

- 示例：某个模拟测试中的实现细节

- 基本数据结构：Atomic*、*Cell、Mutex等等

内部可变性：RefCell

- 本质：使用智能指针将编译期的借用检查延迟到了运行期
- 基本操作
  - borrow、try_borrow：获得不可变引用
  - borrow_mut、try_borrow_mut：获得可变引用
  - 当借用规则不满足时，非try系方法会panic，try系方法会返回错误

- 示例：使用RefCell解决之前的模拟测试中的实现
- 示例：使用Rc+RefCell实现共享所有权+可变引用

并发

Concurrency

并发与并行

- 并发：同时运行多个任务的宏观现象
- 并行：同时运行多个任务的微观事实
  - 并行 包含于 并发；并发 包括 并行
  - 宏观和微观是相对于特定的执行单元的
- Rust的口号：无畏并发——在并发代码中保证不出现数据竞争导致的内存安全问题。
  - 不保证不出现死锁（不是内存安全问题）！
- How?
  - 本质上通过Send、Sync两个标记特征。

进程与线程

- 对于操作系统内核来说
- 进程：资源分配的基本单位
- 线程：任务执行的基本单位

- 多线程：将一批任务分配给多个线程同时处理
  - 目的：并发

使用多线程

- 示例：使用std::thread创建线程，使用JoinHandle等待线程完成
- 示例：使用move关键字进行多线程的闭包参数捕获

多线程数据访问：消息传递

- std::mpsc（Multiple Producer Single Consumer）通道
  - 使用std::mpsc::channel创建通道对象
  - 使用send发送数据
  - 使用recv接收数据
  - 使用drop关闭通道
- 示例：mpsc通道的使用

多线程数据访问：状态共享

- Mutex（MUTual EXclusion）单次互斥访问
  - 使用lock获得互斥锁（LockGuard）
  - 对LockGuard使用Deref（Mut）特征解引用访问数据
  - 使用drop释放互斥锁
- 示例：使用Mutex共享数据

多线程相关特征：Send、Sync

- Send：数据可安全地在线程间传递
- Sync：数据可安全地在线程间共享
- T: Sync ⇔ &T: Send
- 二者是unsafe特征，在手动实现的时候需要加unsafe关键字
- 二者是auto特征，可以通过结构化推导自动实现

- 示例：查看thread::spawn定义中的Send特征
- 示例：检查常用数据结构中的Send和Sync特征
- 示例：使用Arc智能指针

感谢聆听

Thanks for listening