链滴

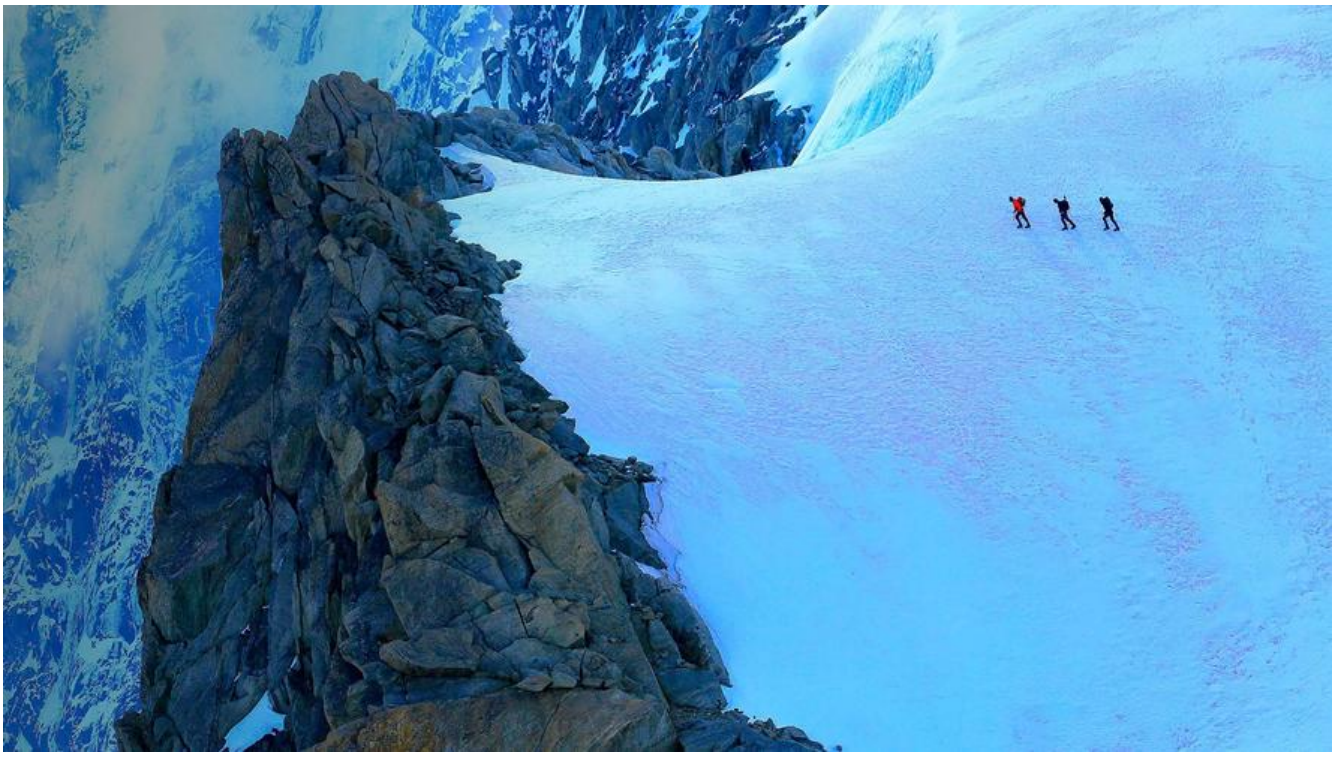
阅读 todos-iced 示例代码

作者：plus7wist

原文链接：https://ld246.com/article/1618287519024

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



todos-iced 是一个 iced 项目的示例，它实现了待办管理，并且持久化待办任务到一个 JSON 文件里。

iced 是一个 GUI 框架，卖点是跨平台、Rust、受 Elm 启发。Elm 是一门设计成编译产出 Javascript 的函数式编程语言，它特化为要解决 GUI 问题。它有 view、update、model 三部分设计，这都被 ic d 吸收了。

todos-iced（下面简称 todos）（目前）代码有六百多行，是中型的示例，涵盖了很多 iced 的功能 阅读 todos 有助于了解如何使用 iced 开发 GUI 工具。

主函数和 Application

主函数里只有 Todos::run(Settings::default()) 一行，返回的是 iced::Result，其中的 Settings 是 ice 提供的，既然使用了默认值，那么忽略它即可。Todos 是 todos 定义的类型，它是整个应用的核心 象。但 run 函数并不是 todos 定义的，而是因为 Todos 实现了 iced::Application，后者提供了这个 数。

Todos 是：

```
enum Todos {
    Loading,
    Loaded(State),
}
```

todos 启动时会从文件系统里加载之前保存的 todos.json，这里保存了 todos 的状态。从启动到加 结束这个过程中，todos 处在 Loading 状态，之后就处于 Loaded 状态，Loaded 状态里自然就获 了 todos 的状态信息，用 State 表示。State 之后再解析其内容。

Todos 对 Application 的实现是应用的核心。这个实现绑定了三个关联类型：

```
type Executor = iced::executor::Default;
type Message = Message;
type Flags = ();
```

第一第三条类型都是很平凡的类型，所以略过不看。第二条的 Message 则是 todos 实现的类型，它一个 enum，输入事件产生的消息都是 Message 类型（的变体）。

MVC（模型、视图、控制器）的经典设计是，输入事件产生消息（Message），应用（Application 会处理消息，更新用户可以看到的图形（视图），这样用户就认为自己的输入事件（鼠标点击、键盘入）产生了响应。

Application 还实现了以下函数：

```
fn new(_flags: ()) -> (Todos, Command<Message>)
fn title(&self) -> String
fn update(&mut self, message: Message, _: &mut Clipboard) -> Command<Message>
fn view(&mut self) -> Element<Message>
```

下面会详细介绍这些函数。

Application 的 new 函数和 title 函数

new 显然是创建 Todos的函数，只是同时还返回了Command<Message>。Command 是 iced 的部分，Message 则是上面提到的消息类型。这个函数只有一句：

```
fn new(_flags: ()) -> (Todos, Command<Message>) {
    (
        Todos::Loading,
        Command::perform(SavedState::load(), Message::Loaded),
    )
}
```

应用启动的时候处于 Loading 状态我们已经知道了。Command::perform 的第一个参数其实是一个uture，第二个参数是 Message 的一个变体构造器，这个变体有一个未命名的字段，所以变体名字是一个函数。而且，第一个 Future 的结果正是第二个函数的参数类型，第二个函数的返回值是一条息。也就是说好似在某个异步上下文里执行了：

```
let result = SavedState::load().await;
return Message::Loaded(result);
```

用常见的语言描述，这个函数实际上是运行一个 Future，注册一个对 Future 结果的回调函数，回调数返回的消息则会触发消息处理。

具体功能上，SavedState::load() 则是从文件系统加载 todos.json 里包含的应用状态。因为这是一可能耗时比较长的 IO 操作，所以不能将这个动作写在 UI 循环里；具体在此处，不能让 new 函数同等待加载和解析文件，不然我们的 GUI 程序可能要因等待加载文件而卡住。

title 函数比较简单，它返回应用的标题。它也是应用视图的一部分，所以可以在每次更新视图的时候回不同的标题。

Application 的 update 函数

update 是应用处理消息的函数。再看一下这个函数的签名：

```
fn update(&mut self, message: Message, _: &mut Clipboard) -> Command<Message>
```

剪切板参数在这里比较醒目，但是 todos 不使用它，所以我们也不关心这部分。它的 self 是可变引，我们可以根据消息来改变应用的状态，例如说上面说过的加载 todos.json 过程完成之后，我们就

要将应用从 Loading 改变成 Loaded，这个动作就只需要实现：

```
match (self, message) {
    (Todos::Loading, Message::Loaded(state)) => {
        *self = Todos::Loaded(state);
        Command::none()
    }
    _ => todo!(),
}
```

这里的 Command::none() 返回一个什么都不做的 Command。

update 函数还可以也返回 Command::perform，这样就驱动应用再处理新产生的事件（还是用 update 函数）。

我们可以总结 Command 的设计，它描述一种多半是因为 IO 而需要在 UI 循环外面执行的动作，这动作最终会产生一个消息，等待应用继续处理。这种要执行的动作可以退化成什么都不做，这时也就产生消息。

如果你熟悉 Monad，那么可以将 Command<Message> 视作包裹了 Message 的 Monad，update 函数就是定义这种 Monad 的 bind 函数。在 Monad 展开的时候产生消息，同时会更新视图，对视的更新就是 Monad 所屏蔽的副作用（的一部分）。

Application 的 view 函数

view 函数根据应用当前的状态来绘制 UI。它返回的是 Element<Message>，代表对视图的抽象 iced 提供了按钮、输入框等等组件，可以用它们组合成 Element。在 Todos 是 Loaded 时，Element 里含 TextInput：

```
let input = TextInput::new(
        input,
        "What needs to be done?",
        input_value,
        Message::InputChanged,
    )
    .padding(15)
    .size(30)
    .on_submit(Message::CreateTask);
```

这里出现了两个消息变体，第一个是 InputChanged，它注册在 TextInput 的构造器里。第二个是 CreateTask，它用 on_submit 方法注册。这代表当输入改变时，会产生第一个消息，当（按下回车）提消息的时候会产生第二个事件。注意第一个消息是以 String 为参数的。

对应地在 update 里要实现处理这两个消息的方法，这里只解释第一个消息。当输入框字符串改变了我们需要更新视图，显示用户更新之后的字符串。注意上面的 TextInput 是用 input_value 来构造的它是输入框里显示的字符串，来自 Todos::Loaded 的状态。所以 InputChanged 消息来之后，在 update 里只需要更新 input_value 即可：

```
match (self, message) {
    (Todos::Loaded(_), Message::InputChanged(text)) => {
        self.input_value = text
        return Command::none()
    }
    _ => todo!()
```

```
}
```

这样下次视图更新就会将输入改变的值绘制好了。

更多

还有一些细节没有介绍，主要包括样式设计、字体和其他更多消息的处理。