Ppt1

Rust 标准库中包含一系列被称为 集合（collections）的非常有用的数据结构。大部分其他数据类型都代表一个特定的值，不过集合可以包含多个值。不同于内建的数组和元组类型，这些集合指向的数据是储存在堆上的，这意味着数据的数量不必在编译时就已知，并且还可以随着程序的运行增长或缩小。每种集合都有着不同功能和成本，而根据当前情况选择合适的集合，这是一项应当逐渐掌握的技能。

Ppt2

我们要讲到的第一个类型是 Vec，也被称为 vector。vector 允许我们在一个单独的数据结构中储存多于一个的值，它在内存中彼此相邻地排列所有的值。vector 只能储存相同类型的值。

Ppt3

为了创建一个新的空 vector，可以调用 Vec::new 函数。注意这里我们增加了一个类型注解。因为没有向这个 vector 中插入任何值，Rust 并不知道我们想要储存什么类型的元素。这是一个非常重要的点。现在，所有你需要知道的就是 Vec 是一个由标准库提供的类型，它可以存放任何类型，而当 Vec 存放某个特定类型时，那个类型位于尖括号中。

在更实际的代码中，一旦插入值， Rust 就可以推断出想要存放的类型，所以你很少会需要这些类型注解。更常见的做法是使用初始值来创建一个 Vec，而且为了方便 Rust 提供了 vec! 宏。这个宏会根据我们提供的值来创建一个新的 Vec。因为我们提供了 i32 类型的初始值，Rust 可以推断出 v 的类型是 Vec，因此类型注解就不是必须的。接下来让我们看看如何修改一个 vector。

Ppt4

如果想要能够改变它的值，必须使用 mut 关键字使其可变。放入其中的所有值都是 i32 类型的，而且 Rust 也根据数据做出如此判断，所以不需要 Vec 注解。

Ppt5

我们来看一下如何读取Vector里的元素。一共有两种读取Vector里面值的方式。一种是通过索引，另一种是用get方法。我们看这个例子，这里有两个需要注意的地方。首先，我们使用索引值 2 来获取第三个元素，索引是从 0 开始的。其次，这两个不同的获取第三个元素的方式分别为：使用 & 和 [] 返回一个引用；或者使用 get 方法以索引作为参数来返回一个 Option<&T>。这里看不懂的细节现在不必理解。

Ppt6

Rust 有两个引用元素的方法的原因是程序可以选择如何处理当索引值在 vector 中没有对应值的情况。作为一个例子，让我们看看如果有一个有五个元素的 vector 接着尝试访问索引为 100 的元素时程序会如何处理。

当运行这段代码，你会发现对于第一个 [] 方法，当引用一个不存在的元素时 Rust 会造成 panic。这个方法更适合当程序认为尝试访问超过 vector 结尾的元素是一个严重错误的情况，这时应该使程序崩溃。

当 get 方法被传递了一个数组外的索引时，它不会 panic 而是返回 None。当偶尔出现超过 vector 范围的访问属于正常情况的时候可以考虑使用它。接着你的代码可以有处理 Some(&element) 或 None 的逻辑。例如，索引可能来源于用户输入的数字。如果它们不慎输入了一个过大的数字那么程序就会得到 None 值，你可以告诉用户当前 vector 元素的数量并再请求它们输入一个有效的值。这就比因为输入错误而使程序崩溃要友好的多！

Ppt7

遍历vector中的元素我们需要用到for循环，关于for循环我们现在先大概了解。这里i就是v中的元素，每次会打印出v中一个元素，然后i会自增指向v中下一个元素，最后依次打印出所有元素。由于我们只需要读取v的权力，所以我们这里用了&，关于&我们以后会在涉及。这里只做了解。如果我们要遍历一个可变的vector，我们这里想修改v的值，使得v中的每个元素都加50，那么我们需要用&mut并且在使用 += 运算符之前必须使用解引用运算符（\*）获取 i 中的值。

Ppt1

The Rust standard library contains a series of very useful data structures called collections. Most other data types represent a specific value, but collections can contain multiple values. Unlike the built-in array and tuple types, the data pointed to by these collections is stored on the heap, which means that the amount of data does not have to be known at compile time, and can grow or shrink as the program runs. Each collection has different functions and costs, and choosing the right collection according to the current situation is a skill that should be gradually mastered.

Ppt2

The first type we are going to talk about is Vec, also known as vector. Vector allows us to store more than one value in a single data structure, it arranges all the values next to each other in memory. A vector can only store values of the same type.

Ppt3

To create a new empty vector, you can call the Vec::new function. Note that we have added a type annotation here. Because no value is inserted into this vector, Rust doesn't know what type of element we want to store. This is a very important point. Now, all you need to know is that Vec is a type provided by the standard library, it can store any type, and when Vec stores a specific type, that type is in angle brackets.

In more practical code, once the value is inserted, Rust can infer the type it wants to store, so you will rarely need these type annotations. A more common approach is to use initial values to create a Vec, and Rust provides the vec! macro for convenience. This macro will create a new Vec based on the value we provide. Because we provide the initial value of the i32 type, Rust can infer that the type of v is Vec, so type annotations are not necessary. Next let us see how to modify a vector.

Ppt4

If you want to be able to change its value, you must use the mut keyword to make it mutable. All the values put into it are of type i32, and Rust also makes such judgments based on the data, so Vec annotations are not needed.

Ppt5

Let's take a look at how to read the elements in Vector. There are two ways to read the value in Vector. One is by index, the other is by get method. Let's look at this example, there are two things to note here. First, we use the index value 2 to get the third element, and the index starts from 0. Second, the two different ways to get the third element are: use & and [] to return a reference; or use the get method to return an Option<&T> with an index as a parameter. The details that are not understood here do not need to be understood now.

Ppt6

The reason Rust has two methods of referencing elements is that the program can choose how to deal with the situation when the index value does not have a corresponding value in the vector. As an example, let's see what the program will do if there is a vector with five elements and then try to access the element with index 100.

When you run this code, you will find that for the first [] method, Rust will panic when referencing a non-existent element. This method is more suitable when the program thinks that it is a serious error to try to access an element beyond the end of the vector, and it should crash the program.

When the get method is passed an index outside of the array, it does not panic but returns None. You can consider using it when occasional visits that exceed the range of vector are normal. Then your code can have logic to handle Some(&element) or None. For example, the index may be derived from a number entered by the user. If they accidentally enter a number that is too large, the program will get the None value. You can tell the user the current number of vector elements and ask them to enter a valid value. This is much friendlier than crashing the program due to input errors!

Ppt7

To traverse the elements in a vector, we need to use a for loop. We will first understand about the for loop. Here i is the element in v, one element in v will be printed each time, then i will increment to point to the next element in v, and finally all elements will be printed out in turn. Since we only need the power to read v, we use & here, and we will talk about & in the future. Here is just to understand. If we want to traverse a mutable vector, we want to modify the value of v so that each element in v is increased by 50, then we need to use &mut and must use the dereference operator (\* ) Get the value in i.