

14 集合

Rust 中有很多种集合类型，前面课程内容中，我们见到过一些 `vec![]` 的代码，实际上，这个宏创建一个 `Vec<T>` 类型的值。除了矢量（Vector，这个翻译很奇怪）之外，Rust 还有很多种其他类型的集合，可以和其他语言做个类比：

RUST 集合类型	说明	JAVASCRIPT	JAVA	PYTHON
<code>Vec<T></code>	可变长度的“数组”	<code>Array</code>	<code>ArrayList</code>	<code>list</code>
<code>VecDeque<T></code>	双端队列	-	<code>ArrayDeque</code>	<code>collections.deque</code>
<code>LinkedList<T></code>	双向列表	-	<code>LinkedList</code>	-
<code>BinaryHeap<T></code> <code>where T: Ord</code>	最大堆	-	<code>PriorityQueue</code>	<code>heapq</code>
<code>HashMap<K, V></code> <code>where K: Eq + Hash</code>	键值对哈希表		<code>HashMap</code>	<code>dict</code>
<code>BTreeMap<K, V></code> <code>where K: Ord</code>	有序键值对表	<code>Map</code>	<code>TreeMap</code>	-
<code>HashSet<T></code> <code>where T: Eq + Hash</code>	无序哈希集合	-	<code>HashSet</code>	<code>set</code>
<code>BTreeSet<T></code> <code>where T: Ord</code>	有序集合	<code>Set</code>	<code>TreeSet</code>	-

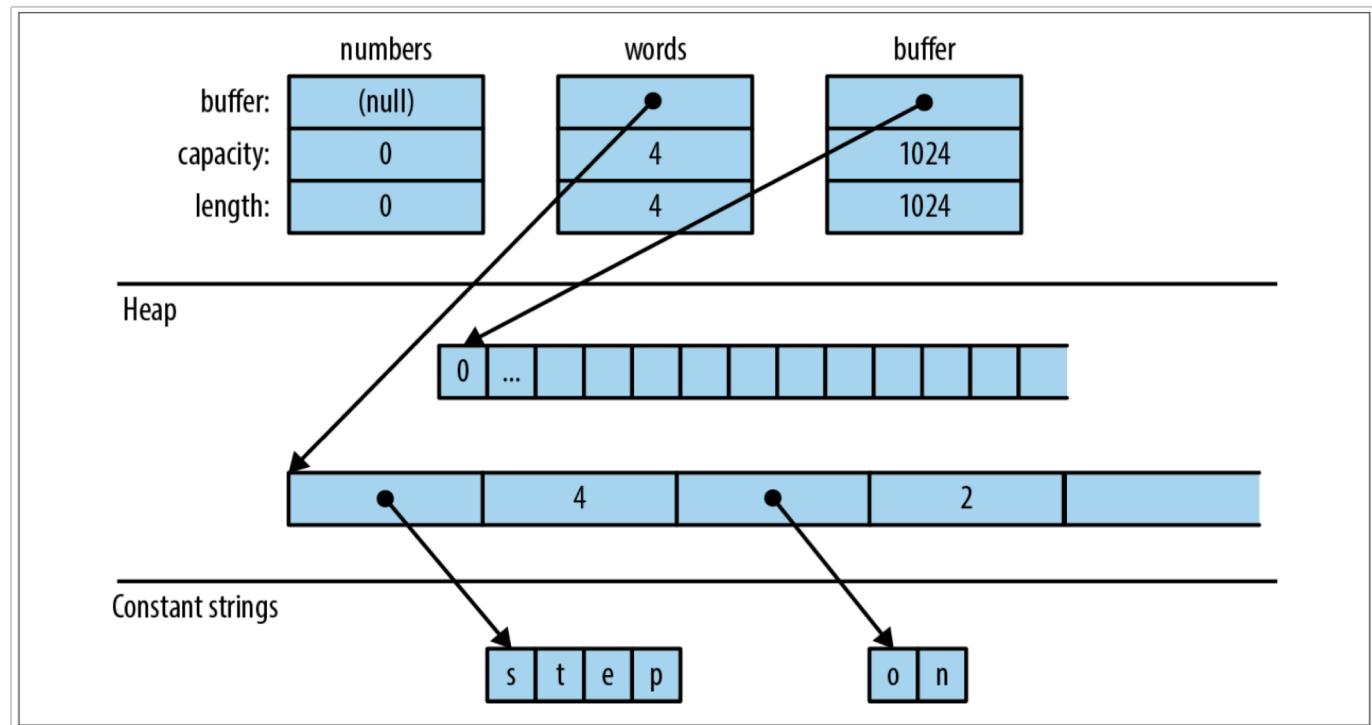
Vec<T>

Rust 提供的宏 `vec![]` 可以让我们方便的构建一个 `Vec` 类型的值：

```

1 fn main() {
2     let numbers: Vec<i32> = vec![];
3     let words = vec!["step", "on", "no", "pets"];
4     let mut buffer = vec![0u8; 1024];
5 }
```

上面示例代码中，行 2 创建了一个空 Vector，由于我们没有指明 Vector 中的元素值，所以 Rust 没有办法推断 `empty_items` 的类型，所以我们需要使用 `numbers: Vec<i32>` 显式指明 Vector 的类型；行 3 在定义 Vector 的同时设置了其中的值；行 4 的写法类似于定义数组，它指明了 Vector 的数据类型、默认值和 Vector 长度，也就是手，Vector `buffer` 中有 1024 个元素，每个元素的值都被设置为 `0u8`。上面示例代码对应的栈和堆的内存布局如下图所示：



从上图可以看出，一个 Vector 的值在栈上包含了：

1. 堆上地址指针，指向 Vector 中的第一个元素（如果有的话）
2. 容量 (`capacity()`)。至当前在堆上申请到连续内存最多可以容纳多少值。可以使用 `Vec::with_capacity(capacity: usize)` 来制定 Vector 的容量
3. 长度 (`len()`)。实际元素的数量

访问集合中的值可以使用下标方式：

```

1 // 获取一个元素的引用
2 let first_line = &lines[0];
3 // 获取一个可 Copy 的元素的拷贝
4 let fifth_number = numbers[4];
5 // 获取一个可 Clone 的元素的克隆
6 let second_line = lines[1].clone();
7 // 获取切片引用
8 let my_ref = &buffer[4..12];
9 // 将切片转换成一个集合，需要元素实现 Clone
10 let my_copy = buffer[4..12].to_vec();

```

除了使用下标方式访问集合中的元素是很直观的，也是很容易理解的。但是很容易造成下标越界的错误。在 JavaScript 中，如果我们访问数组中一个不存在的下标，会得到 `undefined`，程序执行不会受到影响。但是在 Rust 中，如果直接访问集合中一个不存在的下标，会引发错误，程序执行中止。所以，通常情况下，会使用集合提供的方法来访问集合中的元素，避免出现下标越界的异常。集合常用的方法有：

- `first() → Option<&T>` 获取集合第一个元素的只读引用
- `first_mut() → Option<&mut T>` 获取集合第一个元素的可写引用
- `get(index) → Option<&I as SliceIndex<[T]>>>` 获取集合一个或者一段元素的切片引用
- `get_mut(index) → Option<&mut <I as SliceIndex<[T]>>>` 获取集合一个或者一段元素的可写切片引用
- `last() → Option<&T>` 获取集合最后一个元素的只读引用
- `last_mut() → Option<&mut T>` 获取集合最后一个元素的可写引用

如果你确信代码中不存在下标越界的问题，那么访问集合的切片的时候，使用 `[from .. to]` 下标访问的方式也是非常常见的。

遍历 Vector 的方法有：

- `for v in vec` Vector 中的值都被移出，发生所有权转移，也叫做 Vector 被消费了 (consume)
- `for v in &vec` 以只读引用的方式遍历 Vector 中的值
- `for v in &mut vec` 以可写引用的方式遍历 Vector 中的值

HashMap<K, V>

`HashMap<K, V>` 是无序键值对哈希表，`BTreeMap<K, V>` 是有序键值对哈希表。这里的有序和无序是指，当我们遍历这个哈希表的时候，读取到的元素顺序是否和元素插入到哈希表的顺序一致。JavaScript 中的 `Map` 是有序的，也就是遍历元素的顺序和插入元素的顺序是一致的。但是需要提醒的是，很多语言中的哈希表都是无序的，或者有序无序使用两个不同的类型表示，在使用的时候需要注意，根据实际需求选择。

在 Rust 中创建一个 `HashMap` 类型的值就不像 Vector 这么方便了，它没有对应的宏，所以只能使用常规的方法创建：

```

1 fn main() {
2     let m0 = HashMap::<&str, &str>::new();
3     let mut m: HashMap<&str, &str> = HashMap::with_capacity(10);
4     m.insert("hello", "world");
5     m.insert("you", "are great");
6     m.insert("who", "will be the winner?");
7
8     for (k, v) in m {
9         println!("{} = {}", k, v);
10    }
11 }
```

注意，行 2 的代码演示了在调用类型的静态方法的时候，如何使用“turbofish”。

`HashMap<K, V>` 要求 `K` 是实现了 `Eq` 和 `Hash` 这两个 trait 的。因为哈希表有一个内在的逻辑是：如果 `k1 = k2`（要求实现 `Eq`），那么：`hash(k1) = hash(k2)`（要求实现 `Hash`）。我们在实际应用中，大部分时候，用来做哈希表键的值都是简单类型的，比如：整数、字符串等，这些类型都是实现了 `Eq + Hash` 的，极少情况下我们需要自己实现一个

类型的哈希值计算逻辑。

这里补充一些知识，在 Rust, Java 等语言中，都有这样的一个约束：如果两个值相等，那么他们就应该具有相同的 Hash 值；反之不成立：如果两个值的 Hash 值相等，不要求两个值本身相等。在 JavaScript 未看到相关的说法。

`HashMap<K, V>` 的常用方法有：

- `HashMap::new()` 创建一个哈希表类型的值
- `HashMap::with_capacity(capacity)` 使用指定的容量创建一个哈希表类型的值
- `map.len()` 哈希表的长度，也就是其中包含的键值对的数量
- `map.is_empty()` 哈希表是否为空
- `map.contains_key(&key)` 检测给定的 Key 是否存在
- `map.get(&key)` 根据给定的 Key 获取键值对的值。如果存在，则返回 `Some(v)`，如果不存在则返回 `None`
- `map.get_mut(&key)` 根据给定的 Key 获取键值对的值的可写引用
- `map.insert(key, value)` 插入一组键值对
- `map.append(&mut map2)` 把 `map2` 中的键值对都移动到 `map` 中
- `map.remove(&key)` 移除给定的 Key。返回值是 `Option<V>`
- `map.remove_entry(&key)` 移除给定的 Key。返回值是 `Option<(K, V)>`
- `map.clear()` 清空哈希表

上列大部分方法同样适用于 `BTreeMap<K, V>`。

要遍历哈希表中的条目，可以使用以下方法：

- `for (k, v) in map` 哈希表中的条目所有权转移，用 Rust 的术语来说，就是 `map` 被消费了（consume）
- `for (k, v) in &map` 以只读引用的方式遍历哈希表
- `for (k, v) in &mut map` 以可写引用的方式遍历哈希表。注意，这里的“可写”是指对“值”可写。哈希表的“键”一旦进入到哈希表，就不可变

下一节我们会讲解使用迭代器的方式遍历哈希表。

HashSet<T>

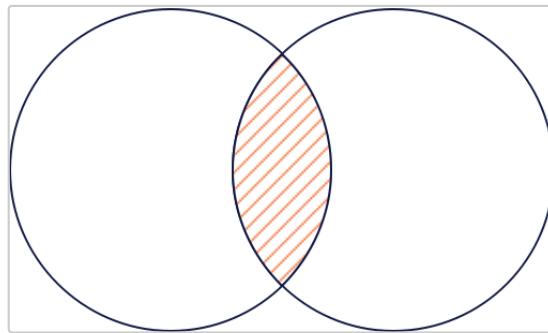
Rust 中的 `HashSet<T>` 是无序的，`BTreeSet<T>` 是有序的。JavaScript 中的 `Set` 是有序的，注意这里的区别。

哈希集合用来存储一组值，这些值在集合中都是唯一的，不可重复。哈希集合中的值也是要求实现了 `Eq + Hash trait` 的。它的常用方法有：

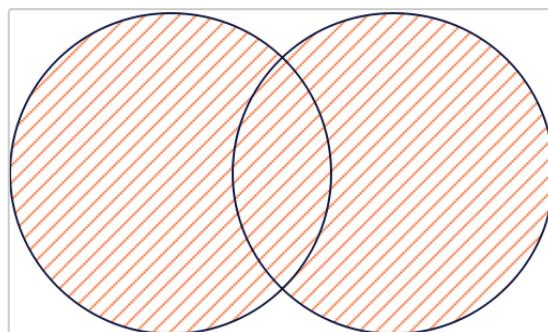
- `HashSet::new` 创建一个哈希集合类型的值
- `HashSet::with_capacity(capacity)` 使用给定的容量创建一个哈希集合类型的值
- `set.len()` 哈希集合的长度
- `set.is_empty()` 检测是否为空
- `set.contains(&value)` 检测是否包含给定的值
- `set.insert(value)` 插入一个值
- `set.remove(&value)` 移除一个值
- `set.get(&value)` 获取一个和给定值相等的值
- `set.take(&value)` 获取并转移所有权
- `set.replace(value)` 替换一个现有的值

哈希集合除了有上面这些常用的方法之外，还提供了一些用于计算多个集合关系的方法，非常方便。为了便于理解，我们采用图形化的方式来表达：

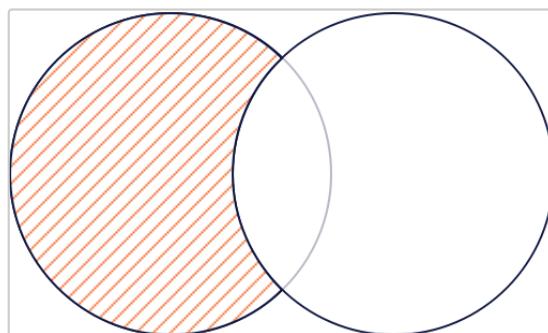
`set1.intersection(&set2)` 交集，同时存在于 `set1` 和 `set2` 中的元素。也可以写作：
`&set1 & &set2`。



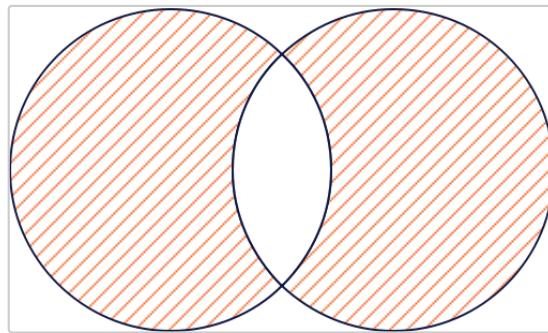
set1.union(&set2) 并集，要么在 set1 中的，要么在 set2 中的元素。也可以写作:
`&set1 | &set2`。



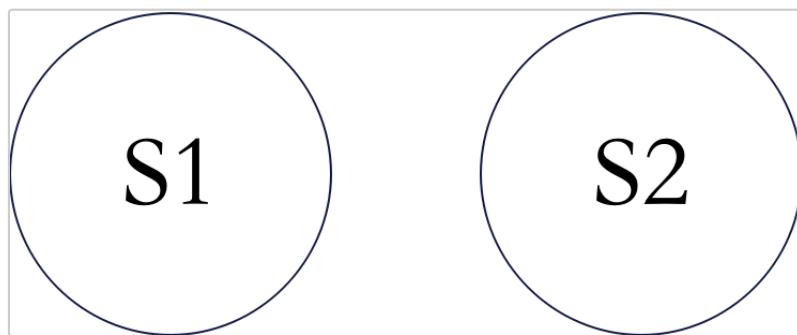
set1.difference(&set2) 差集，在 set1 中但是不在 set2 中的元素。也可以写作:
`&set1 - &set2`。



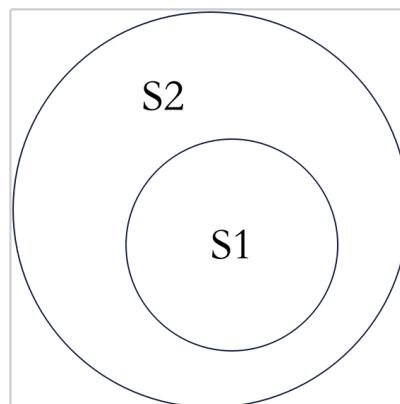
set1.symmetric_difference(&set2) 对称差集，在 set1 或者 set2 中，但是没有同时出现在 set1 和 set2 的。也可以写作：`&set1 ^ &set2`。



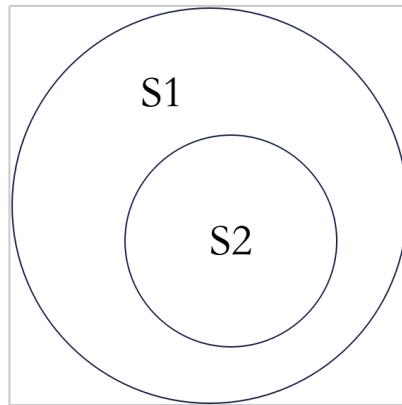
`set1.is_disjoint(&set2)` 判断 `set1` 和 `set2` 是否是不相交的。



`set1.is_subset(&set2)` 判断 `set1` 是否是 `set2` 的子集。



`set1.is_superset(&set2)` 判断 `set1` 是否是 `set2` 的超集。



类似的，`HashSet<T>` 的很多方法也适用于 `BTreeSet<T>`。

要遍历 `HashSet<T>`，可以：

- `for v in set` 遍历并消费哈希集合
- `for v in &set` 以只读引用的方式遍历哈希集合

注意，`HashSet<T>` 不支持以可写引用的方式遍历。这是因为在遍历的过程中如果修改一个值，会导致集合的变化。