Bonus operations for C++/WinRT iterators: The IIterable, and C++/WinRT collections

devblogs.microsoft.com/oldnewthing/20201001-00

October 1, 2020

Raymond Chen

Last time, we saw that C++/WinRT provides a few bonus operators to the `IIterator<T>` in order to make it work a little bit more smoothly with the C++ standard library.

Today we're going to look at `IIterable<T>`, which is an interface that says "You can get an iterator from me."

These Windows Runtime interfaces correspond to analogous concepts in many programming languages.

C++/WinRT	`IIterable<T>`	`IIterator<T>`
	`iterator = o.First();` `for (auto&& v : o) ...`	
C++	`begin`/`end`	`iterator`
	`iterator = begin(o);` `for (auto&& v : o) ...`	
C#	`IEnumerable<T>`	`IEnumerator<T>`
	`enumerator = o.GetEnumerator();` `foreach (var v in o) ...`	
Java	`Iterable<T>`	`Iterator<T>`
	`iterator = enumerable.iterator();` `for (var v : o) ...`	
JavaScript	`@@iterator`	(unnamed)
	`iterator = o[Symbol.iterator]();` `for (v in o) ...`	

As I noted in the table above, these iterators are designed primarily for use by ranged for statements.

```
for (auto&& value : collection)
{
    /* do something with value */
}
```

They can also be used in more general algorithms:

```
std::vector<int> to_vector(IIterable<int> const& collection)
{
  std::vector<int> v;
  std::copy(begin(collection), end(collection), std::back_inserter(v));
  return v;
}
```

Here's a peek behind the scenes: For collections which support a `GetAt` method (such as `IVector`, `IVectorView`, and `IBindableVector`), this is implemented by an internal `fast_iterator`, and the expansion of the ranged for loop comes out like this:

```
auto&& range = collection;
auto size = range.Size();
for (uint32_t index = 0; index < size; ++index)
{
  auto&& value = range.GetAt(index);
  /* do something with value */
}
```

The temporary `range` is part of the ranged for statement. There are some pre-existing subtleties here, which I leave you to learn about.

For collections which are not indexable, but which are nevertheless iterable, the code falls back to the traditional `Iterator`-based loop:

```
for (auto iterator = as_cpp_iterator(collection.First()); iterator; ++iterator)
{
  auto&& value = *iterator;
  /* do something with value */
}
```

That version takes advantage of iterator overloads we saw last time.

But wait, we're not done yet. There's a little gotcha here that we'll look at next time.

Raymond Chen

Follow