# Changing a loop into a promise or task chain

devblogs.microsoft.com/oldnewthing/20160226-00

February 26, 2016

Raymond Chen

If you are dealing with PPL Tasks in C++ or Promises in JavaScript, you run into the problem of having to rephrase loops in the form of callbacks. (On the other hand, if you're using Tasks in C#, then you have the wonderful `await` keyword to do this all for you. If you're a JavaScript programmer, you can look at the async keyword coming to ES7. If you are using C++ resumable functions, then you can use `__await` . More about resumable functions. Still more. )

Let's convert a loop into a promise/task chain. Here's the loop:

```
std::vector<std::unique_ptr<Thing>> things;

void FrobEachThing()
{
    for (auto thing : things) {
       thing->FrobAsync();
    }
}
```

The problem is that the `FrobAsync` method is asynchronous and returns a task, and we want to perform each frob operation in series, not in parallel. If we were writing in C#, this would be a piece of cake:

```
async Task FrobEachThingAsync()
{
    foreach (var thing in things) {
        await thing.FrobAsync();
    }
}
```

Similarly, if we had resumable functions:

```
task<void> FrobEachThingAsync()
{
    for (auto thing : things) {
       __await thing->FrobAsync();
    }
}
```

But we don't have that, so we will need to do the transformation ourselves.

At this point, you think back to Computer Science class and that stuff you learned about recursion and you wondered when anybody would ever want to do that. Well, we're going to do that.

The idea is that we start the asynchronous operation, and pass as a callback a function that knows how to continue the loop. Since this is a loop, the callback may end up passing itself as the next callback, and that's where we get the appearance of recursion. (It's not really recursion because the creation of the task returns immediately; the callback runs when the task completes, which is some time later.)

First, let's write out what that ranged for loop really means:

```
void FrobEachThing()
{
    auto first = begin(things);
    auto last = end(things);
    while (first != last) {
        (*first)->FrobAsync();
        first++;
    }
}
```

With this formulation, it's easier to see how to make it recursive. Actually, the important thing is that you make it *tail-recursive*.

```
typedef decltype(begin(things)) thing_iterator;

void FrobTheRestOfTheThings(
    thing_iterator first,
    thing_iterator last)
{
    if (first != last) {
        (*first)->FrobAsync();
        FrobTheRestOfTheThings(first + 1, last);
    }
}

void FrobEachThing()
{
    FrobTheRestOfTheThings(begin(things), end(things));
}
```

Now that we have tail recursion, we can make it into a task chain:

```
task<void>
FrobTheRestOfTheThingsAsync(
    thing_iterator first,
    thing_iterator last)
{
    if (first != last) {
        return (*first)->FrobAsync().then([first, last]() {
            return FrobTheRestOfTheThingsAsync(first + 1, last);
        });
    } else {
        return create_task([](){}); // null task - base case of recursion
    }
}

task<void> FrobEachThingAsync()
{
    return FrobTheRestOfTheThingsAsync(begin(things), end(things));
}
```

The same logic applies to JavaScript. Starting with

```
function frobEachThing()
{
    things.forEach(function(thing) { thing.Frob(); });
}
```

First, do the rewrite into an explicit loop.

```
function frobEachThing()
{
    var i = 0;
    while (i < things.length) {
        things[i].frob();
        i++;
    }
}
```

Then apply the same logic as above to convert this into a promise chain:

```
function frobTheRestOfTheThingsAsync(array, index, length) {
    if (index != length) {
        return array[index].frobAsync().then(function() {
            return frobTheRestOfTheThingsAsync(array, index + 1, length);
        });
    } else {
        return WinJS.Promise.wrap(); // null task - base case of recursion
    }
}

function frobEachThingAsync()
{
    return FrobTheRestOfTheThingsAsync(things, 0, things.length);
}
```

JavaScript captures by reference and uses garbage collection, so things are a bit easier. We can make one function local to the other and let the closures capture our state.

```
function frobEachThingAsync()
{
    var array = things;
    var length = array.length;
    var index = 0;

    function rest() {
        if (index != length) {
            return array[index].frobAsync().then(function() {
                index++;
                rest();
            });
        } else {
            return WinJS.Promise.wrap(); // null task - base case of recursion
        }
    }

    return rest();
}
```

**Bonus reading**: How to put a PPLTasks continuation chain into a loop.

Raymond Chen

**Follow**