RefPtr Basics

History

Many objects in WebKit are reference counted. The pattern is that classes have member functions named `ref` and `deref` that increment and decrement a reference count. When the `deref` function is called on an object with a reference count of 1, the object is deleted. Many classes in WebKit implement this pattern by deriving from the `RefCounted` class template.

Back in 2005, we discovered that there were many memory leaks, especially in HTML editing code, caused by misuse of `ref` and `deref` calls. We decided to use smart pointers to mitigate the problem. Early experiments showed that smart pointers led to additional manipulation of reference counts that hurt performance. For example, for a function that took a smart pointer as an argument and returned that same smart pointer as a return value, passing the parameter and returning the value would increment and then decrement the reference count two to four times as the object moved from one smart pointer to another.

We solved that problem in 2005 with a set of smart pointer class templates. C++ move semantics, introduced in C++11, made it possible to streamline those class templates without reintroducing reference count churn.

Later, in 2013, we noticed that our use of pointers in general, and smart pointers in particular, was causing a proliferation of null checks and uncertainty about what can be null. We started using references rather than pointers wherever possible in WebKit code.

Maciej Stachowiak created the class template, `RefPtr`, that implements WebKit's intrusive reference counting, and we have since adapted it so that it works well with move semantics. Andreas Kling created a related class template, `Ref`, which works with RefPtr and provides clarity and even greater efficiency when dealing with reference counted objects in contexts where there is no need for a null pointer.

Raw Pointers

When discussing smart pointers such as the `RefPtr` class template we use the term raw pointer to refer to the C++ language's built in pointer type. Here's the canonical setter function, written with raw pointers:

```
// example, not preferred style

class Document {
    ...
    Title* m_title { nullptr };
}

Document::~Document()
{
    if (m_title)
        m_title->deref();
}

void Document::setTitle(Title* title)
{
```

```
    if (title)
        title->ref();
    if (m_title)
        m_title->deref();
    m_title = title;
}
```

RefPtr

`RefPtr` is a smart pointer class template that calls `ref` on incoming values and `deref` on outgoing values. `RefPtr` works on any object with both a `ref` and a `deref` member function. Here's the setter function example, written with `RefPtr`:

```
// example, not preferred style

class Document {
    ...
    RefPtr<Title> m_title;
}

void Document::setTitle(Title* title)
{
    m_title = title;
}
```

Functions that take ownership of reference counted arguments can lead to reference count churn.

```
// example, not preferred style

RefPtr<Title> untitledTitle = titleFactory().createUniqueUntitledTitle();

document.setTitle(untitledTitle);
```

The title starts with a reference count of 1. The `setTitle` function stores it in the data member, and the reference count is incremented to 2. Then the local variable `untitledTitle` goes out of scope and the reference count is decremented back to 1.

The way to define a function that takes ownership of an object is to use an rvalue reference.

```
// preferred style

class Document {
    ...
    RefPtr<Title> m_title;
}

void Document::setTitle(RefPtr<Title>&& title)
{
    m_title = WTF::move(title);
}

…

RefPtr<Title> untitledTitle = titleFactory().createUniqueUntitledTitle();

document.setTitle(WTF::move(untitledTitle));
```

The title makes it all the way into the data member with a reference count of 1; it's never incremented or decremented.

Note the use of `WTF::move` instead of `std::move`. The WTF version adds a couple of compile time checks to catch common errors, and should be used throughout the WebKit project in place of

`std::move`.

Ref

`Ref` is like `RefPtr`, except that it acts like a reference rather than a pointer; it doesn't have a null value.

`Ref` works particularly well with return values; it's often straightforward to be sure that a newly created object will never be null.

```
// preferred style

Ref<Title> TitleFactory::createUniqueUntitledTitle()
{
    return createTitle("untitled " + m_nextAvailableUntitledNumber++);
}
```

Using `Ref` helps makes it clear to the caller that this function will never return null.

Mixing with Raw Pointers

The `RefPtr` class mixes with raw pointers much as the smart pointers in the C++ standard library, such as `std::unique_ptr`, do.

When using a `RefPtr` to call a function that takes a raw pointer, use the `get` function.

```
printNode(stderr, a.get());
```

With a `Ref`, the `get` function produces a raw reference, and the `ptr` function produces a raw pointer.

```
printString(stderr, a.get().caption());
printNode(stderr, a.ptr());
```

Many operations can be done on a `RefPtr` directly, without resorting to an explicit `get` call.

```
Ref<Node> a = createSpecialNode();
RefPtr<Node> b = findNode();
Node* c = getOrdinaryNode();

// the * operator
*b = value;

// the -> operator
a->clear();
b->clear();

// null check in an if statement
if (b)
    log("not empty");

// the ! operator
if (!b)
    log("empty");

// the == and != operators, mixing with raw pointers
if (b == c)
    log("equal");
if (b != c)
    log("not equal");

// some type casts
RefPtr<DerivedNode> d = static_pointer_cast<DerivedNode>(d);
```

Normally, `RefPtr` enforces a simple rule; it always balances `ref` and `deref` calls, guaranteeing a programmer can't miss a `deref`. But in the case where we start with a raw pointer, already have a reference count, and want to transfer ownership the `adoptRef` function should be used.

```
// warning, requires a pointer that already has a ref
RefPtr<Node> node = adoptRef(rawNodePointer);
```

In the rare case where we have a need to transfer from a `RefPtr` to a raw pointer without changing the reference count, use the `leakRef` function.

```
// warning, results in a pointer that must get an explicit deref
RefPtr<Node> node = createSpecialNode();
Node* rawNodePointer = node.leakRef();
```

RefPtr and New Objects

New objects of classes that make use of the `RefCounted` class template are created with a reference count of 1. The best programming idiom to use is to put such objects right into a `Ref` to make it impossible to forget to deref the object when done with it. This means that anyone calling new on such an object should immediately call adoptRef. In WebKit we use functions named create instead of direct calls to new for these classes.

```
// preferred style

Ref<Node> Node::create()
{
    return adoptRef(*new Node);
}

Ref<Node> e = Node::create();
```

Because of the way `adoptRef` is implemented, this is an efficient idiom. The object starts with a reference count of 1 and no code is generated to examine or modify the reference count.

```
// preferred style

Ref<Node> createSpecialNode()
{
    Ref<Node> a = Node::create();
    a->setCreated(true);
    return a;
}

Ref<Node> b = createSpecialNode();
```

The node object is put into a `Ref` by a call to `adoptRef` inside `Node::create`, then passes into `a` and is passed into `b`, all without touching the reference count.

The `RefCounted` class implements a runtime check so we get an assertion failure if we create an object and call `ref` or `deref` without first calling `adoptRef`.

Guidelines

We've developed these guidelines for use of `RefPtr` and `Ref` in WebKit code.

Local Variables

- If ownership and lifetime are guaranteed, a local variable can be a raw reference or pointer.

- If the code needs to hold ownership or guarantee lifetime, a local variable should be a `Ref`, or if it can be null, a `RefPtr`.

Data Members

- If ownership and lifetime are guaranteed, a data member can be a raw reference or pointer.
- If the class needs to hold ownership or guarantee lifetime, the data member should be a `Ref` or `RefPtr`.

Function Arguments

- If a function does not take ownership of an object, the argument should be a raw reference or raw pointer.
- If a function does take ownership of an object, the argument should be a `Ref&&` or a `RefPtr&&`. This includes many setter functions.

Function Results

- If a function's result is an object, but ownership is not being transferred, the result should be a raw reference or raw pointer. This includes most getter functions.
- If a function's result is a new object or ownership is being transferred for any other reason, the result should be a `Ref` or `RefPtr`.

New Objects

- New objects should be put into a `Ref` as soon as possible after creation to allow the smart pointers to do all reference counting automatically.
- For `RefCounted` objects, the above should be done with the `adoptRef` function.
- Best idiom is to use a private constructor and a public `create` function that returns a `Ref`.

Pitfalls

PassRefPtr

Programmers who worked on WebKit before C++11 are familiar with a class template called PassRefPtr (soon to be renamed to DeprecatedPassRefPtr) and you will see it in older WebKit code.

- Any function result or local variable of type PassRefPtr should be replaced with one of type RefPtr or Ref.
- Any argument of type PassRefPtr should be replaced with one of type RefPtr&& or Ref&&.
- Code calling RefPtr::release to turn a RefPtr into a PassRefPtr should instead call WTF::move.

Common mistakes

- Giving a function argument a type of Ref, RefPtr, Ref&&, or RefPtr&& when it should instead be a raw reference or raw pointer. A function that sometimes takes ownership can work just fine with a raw reference or raw pointer. The rvalue reference form is appropriate when passing ownership is the primary way the function is used and is the case that needs to be optimized. Not all setters need to take an rvalue reference.
- Forgetting to call WTF::move can result in unnecessary reference count churn.

Improving This Document

We should add answers to any frequently asked questions are not covered by this document. One or more of the following topics could also be covered by this document.

- copyRef
- releaseNonNull
- How this works when these are stored in collections such as vectors and hash maps.
- Better explanation of when WTF::move is needed and not needed.
- The “protector” idiom, where a local `Ref` variable is used to keep an object alive.
- Perils of programming with `TreeShared`. (Or after we merge `TreeShared` into `Node`, the perils of programming with `Node`).
- Our desire to eliminate `TreeShared` and instead have nodes hold a reference to their first child and next sibling.
- How we we mix reference counting with garbage collection to implement the DOM and the JavaScript and Objective-C DOM bindings.
- Comparison of WebKit intrusive reference counting with other schemes such as the external reference counting in `std::shared_ptr`.
- Guidelines for use of `std::unique_ptr` and `std::make_unique`.
- The `RetainPtr` class template.

If you have any comments on the above or other ideas about improving the clarity, scope, or presentation, please send mail to the WebKit mailing list.