Converting Bind to BindOnce/BindRepeating

jsbell@ - 2017-06-30

Background

Chromium's `base::Bind()` allows the binding of a function and optional arguments to create a base::Callback<> which can later be Run(), possibly with additional arguments. Full docs. This is used extensively in Chromium: posting a task to another thread, registering observers, scheduling operations in a queue, signaling that an async task is complete, etc. A `base::Closure` is a `base::Callback` with no additional arguments required - an extremely common case, often used to signal completion of a task.

Notably, these types did not distinguish between a callback/closure that would be called once (e.g. a posted task, a scheduled operation, a completion signal) or repeatedly (e.g. an observer, a barrier closure). This necessitated copying arguments, and required readers of the code to infer the expected behavior.

In September 2016, tzik@ introduced `BindOnce()/OnceCallback<>/OnceClosure` and `BindRepeating()/RepeatingCallback<>/RepeatingClosure` to distinguish the cases. For now (originally written in July 2017, but still true as of March 2018) the legacy `Bind/Callback/Closure` types are aliases for the "Repeating" variations. Work to convert the code base over to use these new types is ongoing.

Sample conversion CLs

Indexed DB - https://codereview.chromium.org/2941353002/
Cache Storage - https://codereview.chromium.org/2947753002/
Background Sync - https://codereview.chromium.org/2954433002/

Conversion Notes

Basics

`OnceCallback/OnceClosure` are move-only types; the compiler stops you from making accidental copies. Yay! Most of the conversion involves applying these patterns:

Accepting arguments:

> *before:*
> `void DoSomethingAndCallBackWhenDone(const base::Closure& callback) {…}`
>
> *after:*
> `void DoSomethingAndCallBackWhenDone(base::OnceClosure callback) {…}`

Passing callback ownership:

> *before:*
> `runner->PostTask(FROM_HERE, callback);`
>
> *after:*
> `runner->PostTask(FROM_HERE, std::move(callback));`

Binding with callbacks:

> *before:*
> `base::Bind(&Foo::Bar, this, callback);`
>
> *after:*
> `base::BindOnce(&Foo::Bar, this, std::move(callback));`

Storing callbacks:

> *before:*

```
callback_ = callback;
```

after:

```
callback_ = std::move(callback);
```

Invoking callbacks:

before:

```
callback.Run();
```

after:

```
std::move(callback).Run();
```

A function that takes a `OnceCallback` can be called with a `RepeatingCallback`.

This is analogous to calling a function that takes a const parameter with a non-const argument. The signature is the contract for what the function will do with the argument.

This means that most functions can accept a `OnceCallback` even if they are typically called with a `RepeatingCallback`. A common case is when `base::BarrierClosure` is used; this returns a `RepeatingCallback` but the barrier is usually passed to a methods that will each call it exactly once, so the methods should accept a `OnceCallback`.

Work from the "inside out"

It may be tempting to start updating a module by converting calls from `base::Bind` to `base::BindOnce` but this "outside in" usually requires updating the entire module.

A better strategy is to start at the leaf nodes:

1. Convert all instances of `base::Closure` to `OnceClosure/RepeatingClosure`

    - Closures have no arguments, so are the easiest
    - Callers can still use `base::Bind` - repeating callbacks can “narrow” to once callbacks
2. Convert all instances of `base::Callback<>` to `OnceCallback/RepeatingCallback`
    - Slightly more work, and often typedefs/usings are involved.
    - Go typedef by typedef
3. Convert `base::Bind()` to `base::BindOnce()` or `base::BindRepeating()`
    - Do this last.

`PostTask()` is easy

Since PostTask() takes a OnceClosure it’s simple to convert from base::Bind() to base::BindOnce() at the call site.

`PostTaskAndReplyWithResult()` requires symmetry

Although “logically once”, `PostTaskAndReplyWithResult()` require both callbacks to be either `OnceClosure` or `RepeatingClosure` - you can’t mix. If you can’t convert both arguments, you will need to explicitly cast to `OnceCallback`.

Calling APIs that haven’t been updated

You can use `base::AdaptCallbackForRepeating()` to call into an old-style API that takes a `base::RepeatingCallback` but is “logically once” - the API should accept a `base::OnceCallback` but hasn’t been updated yet.

```
void DoSomethingAndCallBackWhenDone(base::OnceClosure callback) {
  LegacyAPI(base::AdaptCallbackForRepeating(std::move(callback)));
}
```

The `RepeatingCallback` returned by `AdaptCallbackForRepeating` will call the wrapped `OnceCallback` on the first invocation and ignore later invocations.

This gets fun in some cases e.g. //net APIs that take a callback and either call it asynchronously or return an error code. This is a common pattern used by callers:

before:

```
net::CompletionCallback callback = base::Bind(&Foo::DidIt, this);
int rv = thing->DoIt(callback);
if (rv != net::ERR_IO_PENDING)
  callback.Run(rv);
```

after:

```
net::CompletionCallback callback =
  base::AdaptCallbackForRepeating(base::BindOnce(&Foo::DidIt, this));
int rv = thing->DoIt(callback);
if (rv != net::ERR_IO_PENDING)
  callback.Run(rv);
```

([Draft document on options for migrating net::CompletionCallback])

Every use of `base::AdaptCallbackForRepeating()` should eventually be replaced ([tracking bug]) by updating APIs to take the appropriate type.

That said, prefer `base::AdaptCallbackForRepeating(base::BindOnce( … ))` to leaving `base::Bind()` around, since the `BindOnce()` captures the semantics of the code.