Explainer: base::OnceCallback

tzik@chromium.org
2016-11-16
Self Link: go/callback-explainer, goo.gl/YZiUL1

base::OnceCallback is a move-only and single-call callback class, that is intended to replace most of base::Callback usage. We added base::RepeatingCallback as an alias of current base::Callback, that can be run more than once.
This document explains the basic usage of base::OnceCallback. See //docs/callback.md for more detail.

Motivations and benefits

Clear lifetime of the bound arguments

Since the internal state of base::Callback is ref-counted, it's unclear when the bound arguments is destroyed. On the example below, |bar| can be destroyed either on the original thread or the destination thread, depending on the thread scheduling.

```
base::Closure closure = base::Bind(&Foo, bar);
task_runner->PostTask(FROM_HERE, closure);
```

With the new code, once the PostTask completes successfully, |bar| will be destroyed on the target thread regardless of the thread scheduling.

```
base::OnceClosure closure = base::BindOnce(&Foo, bar);
task_runner->PostTask(FROM_HERE, std::move(closure));
```

Declaring oneshot-ness in the type

Since the bound arguments are opaque to the caller of Callback::Run(), it's unclear that the callback can be called more than once.

```
void Foo(const base::Closure& closure) {
  closure.Run();
  closure.Run();  // Unsafe. If an object is bound with base::Passed, the second invocation hits CHECK.
}
```

With the new code, oneshot-ness is included in the type. OnceCallback can be called only once, and RepeatingCallback can be called more than once. As OnceCallback requires std::move() on its invocation, the misused case can be found easier as a use-after-move. We may want to use modified clang-tidy or -Wconsumed option to detect the use-after-move statically.

```
void Foo(base::OnceClosure closure) {
  std::move(closure).Run();
  std::move(closure).Run();  // Use-after-move.
}
```

Cleaner movable type support

The implementation of Bind cannot move the bound arguments to the target function by default, since the resulting Callback may be run more than once.

```
void Foo(std::unique_ptr<int>);
base::Bind(&Foo, base::Passed(base::MakeUnique<int>()));  // Needs base::Passed to opt-in to move-out.
base::BindOnce(&Foo, base::MakeUnique<int>());  // OnceCallback moves out the bound arguments by default.
```

Using OnceCallback

Creating a OnceCallback

```
void Foo(int, int) {}
// Bind 123 to the first parameter of Foo.
base::OnceCallback<void(int)> cb = base::BindOnce(&Foo, 123);

void Bar(std::unique_ptr<int>) {}
base::OnceClosure cb = base::BindOnce(&Bar, base::MakeUnique<int>());
```

BindOnce creates OnceCallback. The semantics is mostly same to Bind except for the return type, and all valid Bind arguments are also valid on BindOnce.
Note that BindOnce doesn't need base::Passed on move-only type. It moves the bound arguments from the internal storage to the target function by default.

Running a OnceCallback

```
base::OnceCallback<void(int)> cb;
std::move(cb).Run(42);
// |cb| is consumed by Run(), and no longer valid below.

base::Bind(&Foo).Run("Hello, world!");

base::OnceClosure closure;
base::ResetAndReturn(&closure).Run();
```

Unlike Callback, OnceCallback can be run only via rvalue. Use std::move() or call Run() on a temporary object.

Passing a OnceCallback to another function

```
using CompletionCallback = base::OnceCallback<void()>;
// Take the callback by value.
void DoWorkAsync(CompletionCallback cb) {}

CompletionCallback cb = base::BindOnce(&OnComplete);
// Pass the callback by rvalue-reference.
DoWorkAsync(std::move(cb));
DoWorkAsync(base::BindOnce(&OnComplete));
```

Pass the OnceCallback argument as a rvalue-reference, and take it by value.

Converting Callback to OnceCallback

```
base::Callback<void(int)> cb;
base::OnceCallback<void(int)> cb2 = cb;

void Foo(base::OnceClosure) {}
Foo(cb);
```

Callback is implicitly convertible to OnceCallback, so that consumers of Callback can migrate to OnceCallback before their users migrate.
TODO: Passing OnceCallback to Objective-C blocks

Migration Plan

The migration from Callback to OnceCallback will have several phase.

Migrate threading primitives

As the first phase of the migration, we should migrate base::TaskRunner and its subclasses. Since most of callback objects are eventually consumed by base::TaskRunner, OnceCallback is generally ready to use after this phase.

Migrate trivial part with a clang refactoring tool

Some typical case of base::Bind usage can be replaced with base::BindOnce in bulk using a clang tool. E.g.: implicit conversion from a resulting Callback of Bind to OnceCallback.

Migrate Mojo-generated code

Mojo-generated code is another major consumer of callback objects. We should add a flag to use OnceCallback in the code generator, and migrate the implementation of the interface one by one.

Migrate others

There will remain a number of non-trivial Callback usage that need manual migration. Will write a detailed instruction document for migration, and ask chromium-dev for volunteer.

Discussion:

Q: Can we avoid rvalue qualified Callback::Run()?

Pros:

- std::move(cb).Run() may look unusual
- Violates Google C++ Style Guide, that limits rvalue reference to move-{ctor,assign} and Perfect Forwarding.

Cons:

- Aligns to upcoming C++ standard: bugzilla discussion, N4543, P0045, P0288R1
- Possible compiler use-after-move error detection by clang-tidy or -Wconsumed.
- Uniform usage to both type of Callback, that reduces the complexity of the Callback implementation, and smaller migration cost.

Alternatives:

- Non-const version of Callback::Run() overload.
- Non-const Callback::RunAndReset().
- Non-method base::Run() & pass-by-value.
    - `std::move(cb).Run(arg)` will be `base::Run(std::move(cb), arg)`