base::{Bind,Callback} extension for move-only types and threadunsafe types

cf: Dana's base::Bind and Callback with move-only types and non-threadsafe arguments
Hiroshige's Slide: Thread-Safety around bind()
http://crbug.com/554299, http://crbug.com/561749

2015-12-02
tzik@, hiroshige@
Status: released

Q&A

Goals

- Make Callbacks MoveOnly.
- Make Callbacks OneShot wherever possible.
- Annotate Callback whether it is run in SameSequence or CrossSequence and add more thread-safety checks.
- Migrate incrementally. Annotate the deprecation explicitly in the deprecated names.
- Merge WTF::bind() into Chromium's Bind().
- Remove base::Passed() and base::Owned() (→ std::move()).
- Remove many base::Unretained() and base::ConstRef() by PostTaskAndReplyWithResult with OneShot Callbacks and move-only argument support.

Motivation

Cleaner move-only bound args support by OneShot Callback.

The existing Bind() supports some whitelisted or opted-in move-only types. The user can pass-in a move-only object to a Callback via Passed(), and the Callback passes-out the stored bound argument to the bound function. If the Callback Run()s more than once, it hits a CHECK. So, the Callback is implicitly restricted to be oneshot. There's no way to know a given Callback can Run() more than once.
This proposal eventually removes Passed(), allows the user pass a move-only type directly to Bind() for a strongly typed OneShot Callback, and forbid passing it to Repeating Callback. So that we can remove Passed() and the whitelist of the move-only types, can call a Callback more than once if the Callback type explicitly allows.
Example:

```
void F(const Closure& cb) {
  cb.Run();
  cb.Run();  // Hits CHECK if a bound arg is Passed().
  // No way to know whether |cb| can be called twice.
}

void G(RepeatingClosure cb) {
  cb.Run();
  cb.Run();  // OK. RepeatingCallback doesn't pass-out.
}

void H(OneShotClosure cb) {
  base::Run(std::move(cb));
  base::Run(std::move(cb));  // Use after move of |cb|. The caller must avoid it.
}
```

Predictable bound args lifetime by MoveOnly Callback.

Callback holds the bound arguments in its internal storage, but on existing implementation and when it's used for PostTask(), it's not predictable on which thread the bound arguments are destroyed. That may cause data race, especially on ref-counted objects including WTF::String and base::RefCounted.
This proposal restricts the Callback type of PostTask() to MoveOnly and OneShot, so that PostTask() doesn't leave any reference to the bound args and the lifetime of them gets predictable.
Example:

```
Closure cb = Bind(&Foo, make_scoped_refptr(new T));
task_runner->PostTask(FROM_HERE, cb);
// Racy. |cb| may be destroyed either on the original thread or the target thread.
// If Foo() retains the ownership of the instance of T, the refcount bump happens on both threads.

// CrossSequence implies OneShot and MoveOnly.
```

```
CrossSequenceClosure cb = CrossSequenceBind(&Foo, make_scoped_refptr(new T));
task_runner->PostTask(FROM_HERE, std::move(cb));
// Safe. |cb| is destroyed on the target thread if the PostTask completes successfully.
// Being MoveOnly is important for this safety.
```

Auditing cross thread data passing by CrossSequence/SameThread Callback.

Blink has two variants of bind(): WTF::bind() for same-thread tasks and blink::threadSafeBind() for cross-thread tasks. blink::threadSafeBind() transforms each bound argument to a thread-detached form. So that it can pass thread-unsafe data across threads. Our base::Bind() doesn't have this mechanism, and we'd like to add it.
This proposal introduces SequenceAffinity flag to specify thread safety explicitly in the Callback types. So that we can enforce to use cross-thread-aware version of Bind() on cross-thread PostTask()s, and also we can safely allow to Bind() a Callback into another Callback. Without this flag, we have to deny to Bind() any Callback, since a Callback may contain a thread-unsafe object as a bound argument.
Example:

```
// Assuming T is non-threadsafe RefCounted.
void F(scoped_refptr<T> obj) {
  task_runner->PostTask(FROM_HERE, Bind(&Foo, obj));
  // Racy. The ref-count bump may happen on both thread.
}

// Does something asynchronously, and calls |cb| on the original thread upon the completion.
void G(const Closure& cb) {
  task_runner->PostTask(FROM_HERE, Bind(&Bar, cb));
  // Racy if the caller did Bind() a non-threadsafe object to |cb|.
  // I.e. if the caller of G makes |cb| by binding an thread-unsafe object like:
  //   cb = Bind(&Baz, thread_unsafe_object);
  // G brings the object to another thread together with |cb|.
  // Even if |cb| eventually Run() on the original thread as a completion callback,
  // the destructor of the BindState and bound arguments may run on the other thread,
  // which is a racy operation to the thread-unsafe object.
}

void H(CrossSequenceClosure cb) {
  task_runner->PostTask(FROM_HERE, CrossThreadBind(&Bar, std::move(cb)));
  // OK. The caller took care of the thread-safety of all bound arguments.
}

void J(SameThreadClosure cb) {
  task_runner->PostTask(FROM_HERE, CrossThreadBind(&Bar, RelayToCurrentThread(std::move(cb))));
  // OK. RelayToCurrentThread() wraps a Callback to keep Run() and destruction happen
  // on the original thread.
}
```

Merge base::Bind and WTF::bind.

We have two separate partial application stuff as base::Bind() and WTF::bind() for a historical reason. They have slightly different semantics and needs nontrivial work to merge. base::Bind() has better support to specify the ownership of the bound arguments. WTF::bind() has a variant for cross thread postTask() as blink::threadSafeBind(), and to ensure the thread-safety, the Callback object itself is move-only and mostly oneshot.

New Callback Signatures

```
enum CopyMode {
  MoveOnly,
  Copyable,
};
```

```
enum RepeatMode {
  OneShot,
  Repeating,
};

enum SequenceAffinity {
  SameThread,
  SameSequence,
  CrossSequence,
  Unspecified,
};

template <typename RunType, CopyMode, RepeatMode, SequenceAffinity>
class CallbackImpl;

template <typename RunType>
using Callback =
    CallbackImpl<RunType, MoveOnly, OneShot, SameSequence>;

template <typename RunType>
using RepeatingCallback =
    CallbackImpl<RunType, MoveOnly, Repeating, SameSequence>;

template <typename RunType>
using CrossSequenceCallback =
    CallbackImpl<RunType, MoveOnly, OneShot, CrossSequence>;
```

CopyMode: → RepeatMode: ↓	MoveOnly	Copyable
OneShot	Callback (SameSequence) CrossSequenceCallback (CrossSequence) ThreadUnspecifiedCallback (Unspecified)	(None)
Repeating	RepeatingCallback (SameSequence) ThreadUnspecifiedRepeatingCallback (Unspecified)	LegacyCallback (Unspecified)

We provide three callback classes.
All callbacks are **MoveOnly**: have move-constructors and move assignment-operators. Copy-constructors and copy-assignment operators should explicitly be deleted.

We don't allow arbitrary combination of CopyMode/RepeatMode/SequenceAffinity, and only allows/exposes the six Callbacks: (Red indicates deprecated and to be eventually removed, see "Incremental Migration" below)

Particularly, we **don't** provide Oneshot && Copyable Callback, because we cannot enforce oneshotness, e.g.:

```
CopyableOneShotCallback cb2 = cb;
base::Run(std::move(cb)); // First time.
base::Run(std::move(cb2)); // Second time. Crash!
    // But it looks as if OK because it doesn't violate std::move() sematics.
```

Callback (MoveOnly, OneShot, SameSequence)

OneShot: can be called only once, and to ensure that, it has rvalue-qualified Run method.
So, its invocation will be

```
base::Run(std::move(cb), /* args */);
```

or

```
std::move(cb).Run(/* args */);
```

Calling multiple times leads null ptr access:

```
base::Run(std::move(cb), /* args */);  // First time. OK.
base::Run(std::move(cb), /* args */);  // Second time. Crash!
```

And it will be easy to detect these cases because it violates std::move() semantics.

SameSequence: constructor, Run() and destructor must be executed on the same sequence.
It has DCHECKs to check it.

RepeatingCallback (MoveOnly, Repeating, SameSequence)

Repeating: can be called multiple times, and can be called e.g.

```
base::Run(cb, /* args */);  // First time. OK.
base::Run(cb, /* args */);  // Second time. OK.
```

or,

```
cb.Run(/* args */);  // First time. OK.
cb.Run(/* args */);  // Second time. OK.
```

Also we can move the callback if the repeating callback must no longer be called:

```
base::Run(cb, /* args */);
// Run the target function with moved bound argument and
// invalidate |cb|.
base::Run(std::move(cb), /* args */);
// Crashes if we call base::Run(cb, /* args */); here.
```

SameSequence: required for thread-safety.

CrossSequenceCallback (MoveOnly, OneShot, CrossSequence)

CrossSequence: We can call Run() or destructor on any thread.
This doesn't have DCHECKs for SameSequence constraint,
but instead we'll pose limitations on the arguments that can be bound.
(E.g. initially, we'll hard-code a list of classes to forbid: WebString, RefCounted, etc. Later, we'll make the check more flexible.).

OneShot: required for thread-safety.

Binding move-only arguments and migrating base::Passed()/base::Owned()

Note that binding move-only arguments is different matter from making Callbacks `MoveOnly`.

TODO: describe general principle for binding move-only arguments and wrap(), unwrap(), forwarding-unwrap().
base::Run(move(cb)); -> forwarding-unwrap(). Default: std::move().
base::Run(cb); -> unwrap(). Default: const&.
Bind(): use std::forward().

base::Passed: We'll deprecate and remove base::Passed.
Current code:

```
void f(scoped_ptr<int>);
scoped_ptr<int> p;
Callback cb = base::Bind(&f, base::Passed(&p));
cb.Run();  // Results in a call to f(std::move(cb.p)).
```

For **OneShot** Callbacks, we replace base::Passed() with std::move():

```
void f(scoped_ptr<int>);
scoped_ptr<int> p;
Callback cb = base::Bind(&f, std::move(p));
base::Run(std::move(cb));  // Results in a call to f(std::move(cb.p)).
```

and we forbid base::Passed() uses with OneShot Callbacks by static_assert:

```
Callback cb = base::Bind(&f, base::Passed(&p)); // Compile Error!
```

For **Repeating** Callbacks, we forbid base::Passed()-style std::move().

```
void f(scoped_ptr<int>);
scoped_ptr<int> p;
base::RepeatingBind(&f, std::move(p));  // Compile Error!
```

base::Owned:
Current code:

```
void f(int*);
base::Bind(&f, base::Owned(new int));
```

For **OneShot** Callbacks, base::Owned() + OneShot Callbacks can be rewitten to base::Passed(), which will be replaced with std::move() as described above:

```
void f(scoped_ptr<int>);
base::Bind(&f, make_scoped_ptr(new int));
```

For **Repeating** Callbacks, ~~We may rewrite it to (Plan A):~~

```
void f(const scoped_ptr<int>&);
base::RepeatingBind(&f, make_scoped_ptr(new int));
```

Or we may leave base::Owned() (Plan B):

```
void f(int*);
base::RepeatingBind(&f, base::Owned(new int));
```

Since const ref to scoped_ptr is banned by our style guide, we'll take Plan B.

Discussion Memo:
Plan A:
pros (tzik@):

It works without specialized Unwrap for OwnedWrapper.
It works for other move only types than scoped_ptr.
RepeatingBind(&f, std::move(str)) + f(const string&) will be common, and, IMO, this doesn't require conditional branch on move-only and movable type.

cons (hiroshige@):

Difference between what was base:Passed() and what was base::Owned() is small.
`void f(scoped_ptr<int>)` indicates base::Passed()-style semantics.
`void f(const scoped_ptr<int>&)` indicates base::Owned()-style semantics.
Can we distinguish them like we are distinguish base::Passed() and base::Owned()? Can we maintain these differences? `void f(const scoped_ptr<int>&)` seems too weak as annotation and hard to search.
This difference is important in CrossSequence cases, but in such cases CrossThreadCallback is OneShot, which implies using `void f(const scoped_ptr<int>&)` (i.e. base::Owned()-style semantics) will lead to compile error, so perhaps safe?

Incremental Migration

Bind() has a huge number of callsites (~20,000!!) and thus migration cannot be done in a lockstep.
In addition to three new Callback classes above, we also add three legacy Callback classes and adapters for migration only.

Step 1: Introduce the new Callback classes, aliases and primitives.
Step 2: Replace the existing `Callback` with `LegacyCallback` (which should be compatible to old one) to avoid name conflict with new `Callback`.
This can be done mechanically.
Step 3: Update `LegacyCallback` to new Callback classes.
This is manually done and consumes most of the time.
This step should be done incrementally, i.e. rewriting a part of Bind()/Callback-related code in each CL.

- Replace it with (new) `Callback`, `RepeatingCallback`, or `CrossSequenceCallback` as long as possible.
- If we are not sure about oneshotness and/or sequence affinity, we can also use legacy Callback classes, but they should be eventually replaced.
- To bridge the migrated code and unmodified code, we use conversion functions and adapters to convert Callbacks (See "Conversion and adapter" section).
  Step 3.1: Update primitives such as `Bind()`, `PostTask()`, etc.
  Step 3.2: Update call sites of `Bind()` and `PostTask()` and others.

Step 4: Delete deprecated classes and adapters (indicated by red color). We might delete CopyMode parameter because all Callbacks will be MoveOnly.

Legacy Callback classes for migration

```
template <typename RunType>
using ThreadUnspecifiedCallback =
    Callback<RunType, MoveOnly, OneShot, Unspecified>;
template <typename RunType>
using ThreadUnspecifiedRepeatingCallback =
    Callback<RunType, MoveOnly, Repeating, Unspecified>;
template <typename RunType>
using LegacyCallback =
    Callback<RunType, Copyable, Repeating, Unspecified>;
```

`LegacyCallback` has the same semantics to the existing Callback.

Conversion and adapter

We'll allow type conversions that tighten the flags:

- Repeating → OneShot
- Copyable → MoveOnly
- CrossSequence → SameSequence

And add Unsafe- or Deprecated-prefixed adapters that loosen the flags.
RepeatMode:

- UnsafeRepeatingCallbackAdapter for OneShot → Repeating
  - The second invocation causes crash (assertion failure).

CopyMode:

- DeprecatedCopyableCallbackAdapter for MoveOnly → Copyable

SequenceAffinity:

- UnsafeCrossSequenceCallbackAdapter for SameSequence → CrossSequence
  - Will hit DCHECK if the converted Callback is invoked / destructed on a wrong sequence.
- DeprecatedSameSequenceCallbackAdapter for Unspecified → SameSequence
- UnsafeCrossSequenceCallbackAdapter for Unspecified → CrossSequence
- UnsafeThreadingUnspecifiedCallbackAdapter for {Same,Cross}Sequence → Unspecified

Discussion memo: Is it enough to have UnsafeLegacyCallbackAdapter only that converts any Callback into `LegacyCallback`?

Checking if migration is correct

Some mistakes in CopyMode/RepeatMode/SequenceAffinity are caught at compile time, but others are not at compile time and crashes at run time, if a specific path is executed (see the first 2 cases in the table below). If such paths are tested in the tests, we can fix them locally, but there will be Bind()/Callbacks that are not fully tested and I'm afraid we cannot notice migration errors (and as we'll rewrite 20,000 Bind() call sites, even the error rate of 0.1% corresponds 20 bugs).

We'd propose:

- Use CHECK() if it is canary/dev(or even beta?) and make them to crash, and look at crash server. I expect crashing dev or beta and fixing bugs before stable is better than leaving potential security bugs in stable.
- Such crashes due to assertion failure occurs at Run(). We have to make it possible to get information of the corresponding callsite of Bind() (by FROM_HERE or so?).

Callback should be:	But the actual code is:	Works fine in the wild?	How can we detect?
Repeating	OneShot	N (Crash)	Test crashes if the code path is tested.
CrossSequence	SameSequence	N (Crash by assert, potential security bugs)	
OneShot	Repeating	Yes (but we'd like to reduce such cases for code health)	We can insert logging code locally and detect "RepeatingCallback that aren't called more than once in tests" and "CrossSequenceCallback that aren't called cross sequence in tests". Then we can manually inspect whether they can actually be converted into OneShot/SameSequence.
SameSequence	CrossSequence		
MoveOnly	Copyable	Yes (but for migration only)	Eventually all Copyable and Unspecified are removed.
CrossSequence	Unspecified		
SameSequence	Unspecified		
Copyable	MoveOnly	(No such case)	

Diffs from proposals in dana@'s doc

1. Make a base::SingleUseCallback and base::RepeatingCallback.
Our proposal is basically similar to this. Expressing the intent (OneShot, Repeating, etc.) at the time of bind is very valuable for stricter checks and clearer code. We have multiple Callback types but this is not a problem, as we assign different types to different callbacks that shouldn't be mixed.

Our proposal adds things about thread safety, incremental migration, and other details.

2. Make a base::MoveOnlyCallback that supports both the single-call and repeated-call cases.
We prefer not mixing OneShot and Repeating cases, for code health, and for thread safety (distingiushing these two is important for cross-thread cases).

3. Make a base::MoveOnlyCallback with a Clone() method.
4. Use idea 3 with Clone() to support repeating callbacks.
We assume/hope that we can make all Callback to be move-only. If Copyable/Repeating Callback is absolutely required, we can bring back Copyable Repeating SameSequence Callback. But we omit it from the first step, assuming there's no such case.

Threading

TODO: We expect we should think more about the details of threading.

PostTaskAndReply

Primitives

Classes

- CallbackImpl

Aliases

- `Callback`
- `RepeatingCallback`
- `CrossSequenceCallback`
- `ThreadUnspecifiedCallback`
- `ThreadUnspecifiedRepeatingCallback`
- `LegacyCallback`

Bind

- `Callback</* RunType */> Bind(Runnable&& runnable, BoundArgs&& args)`
- `RepeatingBind()`
- `CrossSequenceBind()`
- `ThreadUnspecifiedBind()`
- `ThreadUnspecifiedRepeatingBind()`
- `LegacyBind()`

PostTask

- `TaskRunner::PostTask(CrossSequenceCallback<void()>);`
- `TaskRunner::PostTask(ThreadUnspecifiedCallback<void()>);`
- `TaskRunner::PostTask(ThreadUnspecifiedRepeatingCallback<void()>);`
- `TaskRunner::PostTask(LegacyCallback<void()>);`
- `TaskRunner::PostTaskAndReply(CrossSequenceCallback<void()> task,`
  `Callback<void()> reply);`
- `TaskRunner::PostTaskAndReplyWithResult(`
  `CrossSequenceCallback<R()> task, Callback<void(R)> reply);`

PostTaskToCurrentSequence

- base::PostTaskToCurrentSequence(Callback<void()>);
  - Not a TaskRunner member function.

RelayToCurrentSequence

- CrossSequenceCallback<RunType> base::RelayToCurrentSequence(Callback<RunType>);

Implementation (wip)

Planning to refactor bind_internal.h.

```
template <typename F>
struct RunnableTraits {
  static constexpr bool is_callback = /* */;
  static constexpr bool is_function = /* */;
  static constexpr bool is_member = /* */;
  using ReturnType = /* */;
  using ArgsList = TypeList</* */>;
};

template <typename T>
struct CallbackParamTraits {
  using StorageType = /* */;
  static const T& Unwrap(const StorageType&);
  static T Unwrap(StorageType&&);
};

// Generic Runnable runner. Specialized to functions, methods, Callback
// and maybe std::function. Also, handles WeakPtr, scoped_ptr,
// scoped_refptr, (maybe) std::weak_ptr, std::shared_ptr, std::unique_ptr.
template <typename Runnable, typename… RunArgs>
typename RunnableTraits<Runnable>::ReturnType
Run(Runnable&&, RunArgs&&...);

// For polymorphism or type erasure.
// The instance is held by Callbacks.
// Optionally RefCounted or ThreadSafeRefCounted, depends on
// CopyMode and SequenceAffinity.
//
// Requirement:
//   * Sanity check for BoundArgs. The first argument of a method must not be
//     an array. Other arguments must not be a pointer to a RefCounted object.
//   * If Runnable is a method and the first argument is a raw pointer,
//     retain its ownership as scoped_refptr<>.
template <RepeatMode, CopyMode, SequenceAffinity,
          typename Runnable, typename... BoundArgs>
struct BindState {
  BindState(Runnable&& runnable, BoundArgs&&...);

  // Only for Copyable. This will be called when a holding Callback is
  // copied.
  void Ref();

  // Called when a holding Callback is reset.
  void Deref();

  // Only for Repeating.
  R Run(UnboundArgs&&... run) const &;

  R Run(UnboundArgs&&... run) &&;

  template <typename T>
  using StorageType = typename CallbackParamTraits<BoundArgs>::StorageType;
```

```
  StorageType<Runnable> runnable_;
  std::tuple<StorageType<BoundArgs>...> state_;

  // MoveOnly → no ref count
  // Copyable & SameSequence → non atomic ref count
  // Copyable & CrossSequence → atomic ref count
  MaybeRefCount<copyability, affinity> ref_count_;

  // SameSequence → thread assertion on Run() and destructor.
  MaybeSequenceChecker<affinity> sequence_checker_;
};

template <typename R, typename… Args,
          CopyMode copy_mode, RepeatMode repeat_mode,
          SequenceAffinity affinity>
class CallbackImpl<R(Args...), copy_mode, repeat_mode, affinity> {
 public:
  CallbackImpl() {}

  // Only for Copyable.
  CallbackImpl(const CallbackImpl&);

  CallbackImpl(CallbackImpl&&);

  R Run(Args...) &&;

  // Only for Repeating.
  R Run(Args...) const;

 private:
  void* bind_state_;
  void (*invoke_)(void*, Args&&...);
  void (*ref_)(void*);  // Only for Copyable.
  void (*deref_)(void*);

  SequenceChecker sequence_checker_;  // Only for SameSequence.
};
```

Discussions

#1: Parameter types on Callback<R(Args...)>::Run()

- **R Run(Args...)**
  Bare args. This causes an extra copy, comparing to direct function call, for pass-by-valued, copyable, non-movable type. std::function uses this signature.
- **template <typename… RunArgs> R Run(RunArgs&&...)**
  Perfect Forwarding. Needs ~150 callsite mods. It's unclear how to chain the Perfect Forwarding beyond indirection around BindState.
- **R Run(std::conditional_t<std::is_move_constructible_v<Args>, Args, const Args&>...)**
  Template magic. Args can not be an incomplete type. Needs special care on define-and-disable-by-static_assert case.

#2: Variadic Function Support

Should we support C style variadic function as follows?

```
base::Callback<int(...)> cb = base::Bind(&printf, "foo: %d");
```

```
cb.Run(42);
```

No. It's technically hard. std::bind() supports it, but std::function doesn't. Unless there's strong demand, it seems not worth supporting even in a simplified version.

#2.5: Data Member Support

Should we support data member access as follows?

```
struct Foo { int bar; };
base::Callback<int(Foo*)> cb = base::Bind(&Foo::bar);
Foo oo = {42};
Run(&oo);  // 42
```

No, though std::function supports this, and it's technichally possible, there seems no good use case.

#3: Any shorthand for moving Callback invocation?

- std::move(cb).Run(/* args */);
  - Looks confusing? → not so.
- base::Run(std::move(cb), /* args */);
  - Too long?
- std::move(cb)(/* args */); // R operator()(Args...)
  - ~~The style guide forbids operator().~~ → Not forbidden.

#4: Alternative name to "SequenceAffinity"?

- SameThread / CrossThread
- SameSequence / CrossSequence
- Sequential / Concurrent
- ~~Synchronous / Asynchronous~~

#5: ~~Can we remove SequenceAffinity and just let Bind variants take care of thread hopping?~~

#6: Can we have Clonable OneShot callback, instead of Copyable OneShot?

#7: Should all callbacks be move-only? Do we have to forbid Copyable OneShot SameSequence Callback?

Pros

- (hiroshige) Easier to understand/use for developers. We don't have to care about whether a callback is MoveOnly or Copyable.
- (tzik) Copyable OneShot Callback will look less clear to be OneShot in its callsite.

Cons

- (tzik) Copyable Repeating Callback may make the migration easier, at least with it we can split the threading annotation phase out of whole migration step.

#8: # of copies and moves on {OneShot,Repeating}Bind and pass-by-{value, cref, rref}.

Let X be a movable type. Following code snippets shows how a object is copied or moved for each receiver and Bind case. For move-only object, only (2), (4), (6), and (10) are valid.

```
void ByValue(X);
void ByCRef(const X&);
void ByRRef(X&&);

// (1) X is copied into BindState and moved out on Run().
```

```
Bind(&ByValue, x);

// (2) X is moved into BindState and will be moved out to the argument
// of |ByValue| without copy. If X is a move-only type, the callee
// has to use this form to retain a mutable one.
// Equivalent to existing base::Passed().
Bind(&ByValue, std::move(x));

// (3) X is copied into BindState and the callee gets a const reference
// to it.
Bind(&ByCRef, x);

// (4) X is moved into BindState and the callee gets a const reference
// to it.
Bind(&ByCRef, std::move(x));

// Forbidden by the styleguide. Similar to ByValue but without move-ctor
// and dtor call.
Bind(&ByRRef, x);  // (5) X is copied into BindState.
Bind(&ByRRef, std::move(x));  // (6)

// (7) X is copied into BindState and copied to pass |ByValue| argument.
RepeatingBind(&ByValue, x);


// (8) X is moved into BindState and copied to pass |ByValue| argument.
// Note that if the callee moves |x| to a variable to store, this is
// as efficient as (10). This is happen on a setter case.
RepeatingBind(&ByValue, std::move(x));

// (9) X is copied into BindState and the callee gets a reference to it.
RepeatingBind(&ByCRef, x);

// (10) X is moved into BindState and callee gets a reference to it.
// Equivalent to existing base::Owned().
RepeatingBind(&ByCRef, std::move(x));

// Compile error. BindState tries to pass stored x to the callee
// as lvalue.
RepeatingBind(&ByRRef, x);  // (11)
RepeatingBind(&ByRRef, std::move(X));  // (12)
```

#9: The 3 Callbacks are expressive enough to migrate from existing Callback?

#10: Doesn't this increase the size of Chrome binary? Can we estimate that?

Adding "virtual" to BindStateBase increase the stripped Chrome size by n MB (n ~ 3).
→ We can implement without "virtual" by sharing the same BindStateBase on both MoveOnly and Copyable Callback. A disadvantage is BindState of MoveOnly Callback needs to have an unused refcount.

#11: Isn’t manual DCHECK_CURRENTLY_ON enough to ensure the threadsafety?

cf: Hiroshige’s BlinkOn slide. No, the error prone part is not the thread misdirection on a PostTask(), but the data race on a bound object, mainly on a thread-unsafe ref-counted object (e.g. WTF::String and base::RefCounted). If such a object is bound to a Callback and the Callback is passed to across the thread, we have to perform special handling to avoid race. For WTF::String case, we make a deep copy of the string to ensure no other object refers to the string. For RefCounted objects, we should at least ensure the refcount is exactly one.

#12: Are most of Bind() actually CrossSequence?

Probably no. There are two major use case of Bind(): (1) a cross thread function call, and (2) a result handler of an asynchronous function. (1) should be a CrossSequence Callback and (2) should be a SameThread or SameSequence.
Unlike to Bind() case, we expect most of PostTask() are cross thread.

Q&A

How about Blink's WTF::bind()?

- Blink's WTF::bind() and blink::threadSafeBind() will be migrated into `Callback` and `CrossSequenceCallback`.