# C++ promises for Chromium

alexclarke@ Status: Canceled (November 2019)
Prototype  Intro Slide Deck  Concerns discussion. Tracking Bug.
http://bit.ly/cpp-promises

## Rationale

Promises are a generalization of PostTask, which improve code health by making it easier to sequence tasks using a standard vocabulary.  They reduce the need for callback and PostTask spaghetti.  Like PostTask they layer on top of base::Callback and use of them is optional. Orthogonally to Promises, traits are our plan for prioritization of tasks.  Currently the base task scheduler supports static priorities but in the future we hope to introduce dynamic prioritization via a new class of traits (this work will be described elsewhere).

There are a great many ways in which a task dependency graph could be expressed.  Perhaps the most natural form for a browser is to use Promises which are part of the web platform. Features implemented in C++ are often specified in terms of promises (e.g. 1, 2). Without library support developers have had to roll their own logic to implement them, which results in complexity (e.g. 1, 2).  Promises provide a uniform API for defining task dependencies which we argue will make the code base easier to read and maintain.  Example Patches.  1, 2, 3, 4.

There are many other examples of code that could be improved by landing promises: e.g. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Promises would provide Mojo users an alternative to adding things like this to deal with the lack of rejection semantics.

```
child_histogram_fetcher->GetChildNonPersistentHistogramData(
    mojo::WrapCallbackWithDefaultInvokeIfNotRun(
        base::BindOnce(&HistogramController::OnHistogramDataCollected,
                       base::Unretained(this), sequence_number),
        std::vector<std::string>()));
```

There are many Mojo uses that could benefit from this due to the fact that we use a callback to receive responses -- if a pipe is broken, no response, no callback; so this is what developers have to do now, creating magic callbacks that if not run, get invoked upon destruction with some default args.

If Mojo messages with responses used promises instead of callbacks, we would no longer have to use the mojo helpers defined in mojo/public/cpp/bindings/callback_helpers.h.

```
// This is a helper utility to wrap a base::OnceCallback such that if the
// callback is destructed before it has a chance to run (e.g. the callback is
// bound into a task and the task is dropped), it will be run the with
// default arguments passed into WrapCallbackWithDefaultInvokeIfNotRun.
// Alternatively, it will run the delete closure passed to
// WrapCallbackWithDropHandler.
```

E.g.

```
foo->Message(mojo::WrapCallbackWithDropHandler(
    base::BindOnce(&Handler, ...),
    base::BindOnce(&Foo::LogError, default_args...));
```

Could be rewritten as:

```
foo->Message()
  .ThenOn(sequence_task_runner, base::BindOnce(&Handler, ...))
  .CatchOn(sequence_task_runner, base::BindOnce(&Foo::LogError, ..., default_args...));
```

It should be no surprise that the exact same issues regarding callback spaghetti have been noticed by the JS community who share many of the same constraints. They went from callback spaghetti to having promises and most recently the ability to await the results of a promise. While there is some overhead to JS generators, the increased clarity of the code can be quite substantial, because instead of callbacks you can have simple straight line code. Ultimately we hope to use the coroutines ts to let us co_await promises when that's officially adopted (probably in C++20) but that should be considered Promises v2.0.

## Alternatives considered

There are alternatives to promises, for example google3 has several mature task graph evaluation systems however those are closed source and would be an even greater departure from the current chromium style than promises would be.

The STL has std::promise and std::future, which are not suitable because rejection is handled via exceptions which are disabled for chromium and they're thread rather than task based primitives which doesn't feel like the right fit for chromium. Also std:promise is designed about obtaining the value rather than running a task when the value is available, which is a problem because for better task scheduling we would like to build a task graph.

## High level design

In general we're aiming for an API that is similar to ES6 Promises to make it easier to implement web specs. It will also make the order of tasks much more explicit, e.g.

```
 ThreadTaskRunnerHandle::Get()
      ->PostTask(FROM_HERE, base::BindOnce([]() { return 1; }))
      .ThenOn(io_task_runner, FROM_HERE,
              base::BindOnce([](int arg) { return arg + 1; }))
      .ThenHere(FROM_HERE, base::BindOnce([](int arg) { ... }));
```

NB promise.ThenHere(...) is a short version of promise.ThenOn(SequencedTaskRunnerHandle::Get(),...)

We also want to make sure the API feels natural for chromium C++ developers, so certain JS idioms won't be used, such as exceptions or passing the resolve and reject executors into the initial promise function. We propose to optionally support promise rejection via the type system. A promise will be defined like so: Promise<ResolveType, RejectType> enabling code like this:

```
Promise<int, std::string> p;
p.ThenHere(
    FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
 .ThenHere(
    FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
 .ThenHere(
    FROM_HERE, base::BindOnce([](int result) -> PromiseResult<int, std::string> {
    // We can reject by returning the reject type here.
    return std::string("Oh no!");
 }))
  // Just like in ES6 a Then() can be passed resolve and reject callbacks.
 .ThenHere(
      FROM_HERE, base::BindOnce([](int result) {
        // We shouldn't get here, the promise was rejected!
      }),
      base::BindOnce([](const std::string& err) {
        EXPECT_EQ("Oh no!", err);
      }));
```

In JS a typical promise is constructed like so:

```
new Promise((resolve, reject)=>{ … }).then(...);
```

In chromium C++ we expect that to look more like this:

```
task_runner->PostTask(FROM_HERE, base::Bind([](){return 123;}))
  .ThenHere(FROM_HERE, [](int result){ /* Do something with result. */ });
```

Unlike C++ exceptions our proposed promise rejection mechanism doesn't require unwind semantics, and it uses constructs allowed by the style guide. ResolveType and RejectType are allowed to be the same, however this requires the use of base::Resolved<> and base::Rejected<> to disambiguate.  E.g.:

```
Promise<int> a;
...
Promise<void, void> b = a.ThenHere(
    FROM_HERE, base::BindOnce([](int result) -> PromiseResult<void, void> {
      if (result % 2 == 0)
        return base::Resolved<void>();
      else
        return base::Rejected<void>();
    });
```

Not all promises need to be able to reject and we support a base::NoReject type which disables CatchOn. For convenience the Promise<> template sets the default value of the RejectType as base::NoReject, allowing Promise<MyType> to represent a promise that can't be rejected.

Currently PostTask returns a bool and accepts only a base::Closure, we propose to change the implementation of PostTask to instead have a templatized Post(Delayed)Task which can accept a base::OnceCallback with any return type.   PostTask will now return a promise for the result of the call back.  The promise handle doesn't have an accessor for the result, but you can observe it via Then/Catch which naturally provides thread safety.

For compatibility with legacy continuation-callback based APIs it will be possible to use a ManualPromise which helps bridge the gap between promises and continuation-callback based code.

```
ManualPromise<int, net::Error> mp(FROM_HERE);
OldApi(mpr.GetResolvedCB(), mp.GetRejectedCB());

mp.promise()
    .ThenOn(
      io_task_runner, FROM_HERE,
      base::Bind([](int result){ /* Do something with |result| on the io thread. */ }))
    .CatchHere(
      FROM_HERE, base::Bind([](net::Error error){/* Do something with the error. */}));
```

We may want to migrate all usages of base::IgnoreResult(TaskRunner::PostTask) to ManualPromise.  E.g.

```
 window_->render_thread_task_runner()->PostTask(
     FROM_HERE,
     base::BindOnce(
         &FakeFunctor::ReleaseOnRT, base::Unretained(this),
         base::BindOnce(
             base::IgnoreResult(&base::SingleThreadTaskRunner::PostTask),
             base::ThreadTaskRunnerHandle::Get(), FROM_HERE,
             std::move(callback))));
```

Would become:

```
ManualPromise<void> mp(FROM_HERE);
window_->render_thread_task_runner()->PostTask(
     FROM_HERE,
     base::BindOnce(
         &FakeFunctor::ReleaseOnRT, base::Unretained(this),
         mp.GetResolveCallback()));
mp.promise().ThenHere(FROM_HERE, callback);
```

## Support for task traits

ThenOn, CatchOn, FinallyOn will all support specifying TaskTraits instead of a task runner, letting you can write code like this:

```
base::PostTask(FROM_HERE, {base::ThreadPool()}, base::Bind(...))
    .ThenOn(FROM_HERE, {content::BrowserThread::UI}, base::Bind(...))
    .ThenOn(FROM_HERE, {content::BrowserThread::IO}, base::Bind(...));
```

## Handling resolve and reject types

A promise will have a resolve type and optionally a reject type with the following signature.

```
template <typename ResolveType>
class Promise;  // Can't reject

template <typename ResolveType, typename RejectType>
class Promise;  // Can reject
```

The Promise type resulting from a PostTask or  ThenHere/ThenOn and CatchHere/CatchOn is inferred based on the callback's return type.  I.e. it's possible for a promise callback (called an executor in ES6) to resolve or reject using the type system.

The return type of a callback can have one of three forms:

1. base::PromiseResult<ResolveType, RejectType>
2. base::Promise<ResolveType, RejectType>

3. Any other type including void.

| **Callback signature** | **Resulting Promise** |
|---|---|
| `T` | `Promise<T>` |
| `PromiseResult<T>` | `Promise<T>` |
| `Promise<T>` | `Promise<T>` |
| `PromiseResult<T, Err>` | `Promise<T, Err>` |
| `Promise<T, Err>` | `Promise<T, Err>` |

Note just like in ES6 promises it's possible for a rejection handler to start a resolution chain. E.g.

```
p.CatchHere(FROM_HERE, base::BindOnce([](net::Error err) { return 123; }))
    .ThenHere(FROM_HERE, base::BindOnce([](int result) { return result+1; }))
    .ThenHere(FROM_HERE, base::BindOnce([](int result) { /* Do something */ }));
```

## PromiseResult template

For ease of use the base::PromiseResult template supports a number of automatic conversions when unambiguous.

| **Type** | **Automatic conversion supported to PromiseResult<ResolveType, RejectType>** |
|---|---|
| base::Resolved<ResolveType> | Yes |
| base::Rejected<RejectType> | Yes |
| base::Promise<ResolveType, RejectType> | Yes |
| ResolveType | Yes if ResolveType != RejectType |
| RejectType | Yes if ResolveType != RejectType |

E.g.

```
base::PromiseResult<int, std::string> Handler() {
  if (foo) return base::Resolved<int>(123);
  if (bar) return base::Rejected<string>("Whooops");
  if (baz) {
    return 123; // Resolve
  } else {
    return std::string("Oh no");  // Reject
  }
}
```

If a callback always returns a promise, this syntax can be used:

```
base::Promise<int, std::string> Handler() {
  return task_runner->PostTask(FROM_HERE, base::BindOnce([](){
    // Do something.
  });
}
```

If a callback never needs to reject then this syntax can be used

```
int ReturnAnswer() {
  return 42;
}
```

## Promise cancellation

We would like to support explicit promise cancellation and prioritization in a later version of the API.  For now same-sequence cancellation is supported via WeakPtr invalidation just like it currently is for PostTask().

NB base::Bind only accepts WeakPtrs for functors returning a void.  **This will prove awkward for promises.**

## Multiple Thens

Sometimes we want to do multiple things in response to a single event. To make the graph more explicit we propose to support multiple thens. E.g. you can do this:

```
Promise<void> p;

p.ThenOn(io_sequence, FROM_HERE, base::Bind(&ThingToDoOnIoSequence)
    .ThenHere(FROM_HERE, base::Bind(&ThingToDoAfterThatOnUiThread);
p.ThenHere(FROM_HERE, base::Bind(&OtherThingToDoOnUiThread)
    .ThenHere(FROM_HERE, base::Bind(&ThingToDoAfterThat);
```

And this:

```
class ServiceProvider {
 public:
  virtual Promise<scoped_refptr<ServiceA>> GetServiceA() = 0;

  virtual Promise<scoped_refptr<ServiceB>> GetServiceB() = 0;

  virtual Promise<scoped_refptr<ServiceC>> GetServiceC() = 0;
};

class ServiceA {
 public:
  static scoped_refptr<ServiceA> Create();
};

class ServiceB {
 public:
  static scoped_refptr<ServiceB> Create(scoped_refptr<ServiceA> a);
};

class ServiceC {
 public:
  static scoped_refptr<ServiceC> Create(scoped_refptr<ServiceA> a,
                                        scoped_refptr<ServiceB> b);
};

class ServiceProviderImpl : public ServiceProvider {
 public:
  void Init() {
    service_a_ = task_runner_->PostTask(FROM_HERE, base::Bind(ServiceA::Create));
    service_b_ = service_a.ThenHere(FROM_HERE, base::Bind(ServiceB::Create));
    service_c_ = Promises::All(service_a, service_b)
       .ThenHere(FROM_HERE, base::Bind(ServiceC::Create));
  }

  Promise<scoped_refptr<ServiceA>> GetServiceA() override {
    return service_a_;
  }

  Promise<scoped_refptr<ServiceB>> GetServiceB() override {
    return service_b_;
  }

  Promise<scoped_refptr<ServiceAC>> GetServiceC() override {
    return service_c_;
```

```
   }

  private:
   Promise<scoped_refptr<ServiceA>> service_a_;
   Promise<scoped_refptr<ServiceB>> service_b_;
   Promise<scoped_refptr<ServiceC>> service_c_;
 };
```

This has a few implications:

1. Promises need to be refcounted.
2. We need to be careful to prevent use after move errors (see below).

## Support for move only types

The ManualPromise and PromiseResult templates use perfect forwarding so nothing special is needed there to support move only types. The code that invokes promise callbacks, however, needs to know if move semantics are required. We propose to conservatively detect this via a process of elimination:

```
template <typename T>
struct UseMoveSemantics {
  static constexpr bool value = !std::is_reference<T>::value &&
                                !std::is_pointer<T>::value &&
                                !std::is_fundamental<std::decay_t<T>>::value &&
                                !IsRefCountedType<T>::value;
};
```

This means we can support Then chains with move only types. E.g.

```
 Promise<std::unique_ptr<int>> p =
      ThreadTaskRunnerHandle::Get()
          ->PostPromise(FROM_HERE,
                        base::BindOnce([]() { return std::make_unique<int>(123); }))
          .ThenHere(FROM_HERE,
                base::BindOnce([](std::unique_ptr<int> result) { return result; }))
          .ThenHere(FROM_HERE,
                base::BindOnce([](std::unique_ptr<int> result) {/* do something */}));
```

## Preventing use after move

The following code is unsafe and it's likely the second callback would segfault. However in debug builds ThenHere/ThenOn and CatchHere/CatchOn will detect if more than one has been registered for a move only type and if so DCHECK.

```
Promise<std::unique_ptr<int>> p = ThreadTaskRunnerHandle::Get()->PostTask(
    FROM_HERE, BindOnce([]() { return std::make_unique<int>(123); }));

p.ThenHere(FROM_HERE,
           BindOnce([](std::unique_ptr<int> i) { // Do something with i. }));


// This will DCHECK
p.ThenHere(FROM_HERE,
           BindOnce([](std::unique_ptr<int> i) { // Do something with i. }));
```

## Why not have OncePromise /  UniquePromise?

A callback can do anything including deleting something so you have no way of knowing if it's safe to call it repeatedly or not. So having distinct OnceCallback and RepeatingCallback makes sense because that documents this.  Promises are different because we already know the type. Consider a Promise<std::unique_ptr<int>> we expect developers to know the semantics of std::unique_ptr<int> it shouldn't come as a surprise that you can't process multiple Thens where the std::unique_ptr<int> is passed by value.

## Preventing uncaught rejection

The JS experience suggests uncaught rejection is likely to be a common problem.  To combat this in debug builds some extra sanity checks are run when a rejectable promise is resolved or rejected to detect the presence required catch handlers in the promise graph. Since we know the base::Location for each promise, we can print a useful error when we DCHECK showing where the promise initially came from and where the leaf node where the possible rejection wasn't handled.

This checking can sometimes be onerous (e.g. in the case of Mojo) so it will be possible to disable it by either constructing a ManualPromise with RejectPolicy::kCatchNotRequired or by Posting a Task with a kCatchNotRequired trait.

## Cross sequence promises

A common pattern in chromium (often via PostTaskAndReply) is to post a task from one sequence which then posts a reply onto the original sequence.  E.g.

```
// On the UI sequence
io_sequence_runner->PostTaskAndReply(FROM_HERE,
                                     base::BindOnce(&ThingToRunOnIoThread),
                                     base::BindOnce(&ThingToRunOnUiThread));
```

The promise equivalent of that would be:

```
// On the UI sequence
io_sequence_runner->PostTask(FROM_HERE, base::BindOnce(&ThingToRunOnIoThread))
    .ThenHere(FROM_HERE, base::BindOnce(&ThingToRunOnUiThread));
```

If you need the continuation(s) to run on another sequence you can do:

```
// On the UI sequence
io_sequence_runner->PostTask(FROM_HERE, base::BindOnce(&ThingToRunOnIoThread))
    .ThenOn(sequence_runner1, FROM_HERE, base::BindOnce(&ThingToRunOnSequence1))
    .ThenOn(sequence_runner2, FROM_HERE, base::BindOnce(&ThingToRunOnSequence2));
```

## Support for Traits

In addition to ThenOn/CatchOn we want to support ThenWithTraits and CatchWithTraits which will allow the scheduler to select the sequence the callbacks run on and to allow for initially static prioritization. Orthogonally to promises we hope to introduce dynamic prioritization via traits (remember traits are inherited by subsequent PostTasks).

## Sequence Safety

Under the hood Then & Catch already use PostTask, so the API is already sequence safe. We will make sure the internal implementation of promise scheduling is sequence safe. Under the hood the promise library uses atomics to maintain parts of the task graph.  This code is relatively simple but will need careful review and testing.

## Interfacing with legacy code

The ManualPromise helps interacting with old style callback APIs. It has methods to return a resolve and reject callbacks.  E.g.

```
base::ManualPromise<void> mpr;
mpr.promise().Then(ui_sequence, FROM_HERE, base::BindOnce(&NextStep));
OldStyleApiWhichRunsStuffOnTheIoThread(mpr.GetResolveCallback());
```

## ThenOn & CatchOn vs ThenHere & CatchHere

For clarity we want it to be obvious which sequence a callback will run on. ThenHere() and CatchHere() are just wrappers around ThenOn() and CatchOn() supplying the current sequence.

## promise.All and promise.Race

ES6 promises support special promises which resolve once one or all promises from an iterable have been resolved. Note If internal google code is anything to judge by, promise.race and promise.all are both used extensively.  Promise.race is typically used to wait for something to happen with a time out, or in UI code to do something when one of N buttons is pressed. Promise.all is typically used to do something when N asynchronous actions have completed, something engineers are already doing via ad hoc methods in chromium.

For error handling clarity we propose require that for All & Race the reject reason must be one of: void/enum/class/struct.  The concern is int or bool make poor error reasons when there's multiple error sources, potentially with different meanings for the same value.

**We won't initially land Promises::Race because we're not convinced it will be necessary in Chromium c++ code.  If we do decide to land it, it will have the following design:**

Promises::Race will support three forms:

1. A vararg list of promises:

```
Promise<int, void> p1;
Promise<void, net::Error> p2;
Promise<void, CustomError> p3;
Promise<int, std::string> p4;
```

```
Promises::Race(p1, p2, p3, p4)
  .ThenOn(sequence, FROM_HERE, base::BindOnce([](Variant<int, Void> result) {
         // Do something with result.
       }),
       base::BindOnce([](const Variant<base::Void, net::Error, CustomError,
                                       std::string>& err) {
         // Do something with err.
       }));

…
p2.Resolve(); // This would trigger the Then above.
```

Note the use of Variant here, std::variant is coming in C++17 and its used to return a value from a set of types.  Depending on when chromium moves to C++17 we will either land a functionally equivalent clone or use std::variant.  For convenience we will deduplicate the types, and only use a variant if there's multiple resolve or reject types.

Note void resolve and reject types require special handling, it's not possible to store a void in std::variant but you can store struct Void{}.

2. A container of Promises:

```
std::vector<Promise<int>> promises;

Promises::Race(promises).ThenOn(sequence, FROM_HERE,
    base::BindOnce([]( int result) {
       ...
    }));
```

3. A container of Variants of Promises:

```
std::vector<Variant<Promise<int, ErrorType1>,
                    Promise<int, ErrorType2>,
                    Promise<void, ErrorType3>>> promises;

Promise<void, ErrorType1> p1;
Promise<int, ErrorType2> p2;
Promise<void, ErrorType3> p3;
promises.push_back(p1);
promises.push_back(p2);
promises.push_back(p3);
Promises::Race(promises)
  .ThenOn(sequence, FROM_HERE, base::BindOnce([](Variant<int, Void> result) {
         // Do something with result.
       }),
   base::BindOnce(
       [](const Variant<ErrorType1, ErrorType2, ErrorType3>& err) {
         // Do something with err.
```

```
              }));
```

Promises::All will likewise support three forms:

1. A vararg list of promises:

```
       Promise<float, ErrorType1> p1;
       Promise<int, ErrorType2> p2;
       Promise<bool, ErrorType3> p3;

       Promise<std::tuple<float, int, bool>> p = Promises::All(p1, p2, p3);
       p.ThenHere(sequence, FROM_HERE,
           base::BindOnce([](std::tuple<float, int, bool> result) {
              // Do something with result.
           }),
           base::BindOnce([](Variant<ErrorType1, ErrorType2, ErrorType3> err) {
              // Do something with err.
           }));
```

std::tuple is awkward to work with, so as syntactic sugar we will support unpacking the tuple like so:

```
       Promise<float> p1;
       Promise<int> p2;
       Promise<bool> p3;

       Promise<std::tuple<float, int, bool>> p = Promises::All(p1, p2, p3);
       p.ThenHere(sequence, FROM_HERE,
           base::BindOnce([](float a, int b, bool c) {
             ...
           }));
```

2. A container of promises:

```
       std::vector<Promise<int>> promises;

       Promises::All(promises).ThenOn(sequence, FROM_HERE,
           base::BindOnce([](std::vector<int> result) {
                    ...
                  });
```

3. A container of variants of promises:

```
       Promise<int> p1;
       Promise<void> p2;
       Promise<std::string> p3;

       std::vector<Variant<Promise<int>, Promise<void>, Promise<std::string>>> promises;
       promises.push_back(p1);
```

```
promises.push_back(p2);
promises.push_back(p3);

Promises::All(promises).ThenHere(FROM_HERE,
    base::BindOnce([](std::vector<Variant<int, Void, std::string>> result) {
      …
    });
```

## Support for finally

This is less needed in C++ but ES6 promises are spec'd with finally callbacks which run after all child resolve & reject callbacks have run. We propose to support this out of completeness and because it's thought some web APIs use this. The runtime overhead of finally is zero if not used, and the binary size implication of using them is low since they're not templatized.  E.g.

```
ManualPromise<void, std::string> p1(FROM_HERE);
Promise<void, std::string> p2 =
    p1.promise()
        .ThenHere(FROM_HERE, BindOnce([]() { LOG(INFO) << "Then #1"; }))
        .ThenHere(FROM_HERE,
                       BindOnce([]()-> PromiseResult<void, std::string> {
                                  LOG(INFO) << "Then #2 (reject)";
                                  return std::string("Whoops!");
                                }))
        .ThenHere(FROM_HERE, BindOnce([]() { LOG(INFO) << "Then #3"; }))
        .ThenHere(FROM_HERE,  BindOnce([]() { LOG(INFO) << "Then #4"; }))
        .CatchHere(FROM_HERE, BindOnce([](const std::string& err) {
                                        LOG(INFO) << "Caught " << err;
                                      }));

p2.FinallyHere(FROM_HERE, BindOnce([]() { LOG(INFO) << "Finally"; }));
p2.ThenHere(FROM_HERE, BindOnce([]() { LOG(INFO) << "Then #5"; }))
p2.ThenHere(FROM_HERE, BindOnce([]() { LOG(INFO) << "Then #6"; }))

p1.Resolve();
```

Will print:

```
Then #1
Then #2 (reject)
Caught Whoops!
Then #5
Then #6
```

```
Finally
```

## Minor ergonomic features

Resolve and Reject results are not always useful, particularly after a Promises::All.  We will optionally support void callbacks.  E.g.

```
Promise<int> p1;
Promise<void, bool> p2;
Promise<void, int> p3;
Promise<int, std::string> p4;

Promises::All(p1, p2, p3, p4)
  .ThenHere(FROM_HERE, base::BindOnce([]() {
         // Do something when done, ignoring individual results.
         DoSomethingOnResolution();
       });
```

For Mojo it's useful for std::tuple unpacking to apply generally. E.g.

```
Promise<std::tuple<float, int, bool>> p;
p.ThenHere(FROM_HERE,
    base::BindOnce([](float a, int b, bool c) {
      ...
    }));
```

### Constructing already resolved or rejected promises

Like ES6 promises we will support construction of already resolved and rejected promises via Promise<>::CreateResolved and Promise<>::CreateRejected.  One common use for this will be default values for client interfaces and another will be for mocking:

```
class ServiceProviderForTest : public ServiceProvider {
 public:
  Promise<scoped_refptr<ServiceA>> GetServiceA() {
     return Promise<scoped_refptr<ServiceA>>::CreateResolved(
         MakeRefCounted<MockServiceA>());
  }

  virtual Promise<scoped_refptr<ServiceB>> GetServiceB() {
     return Promise<scoped_refptr<ServiceB>>::CreateResolved(
         MakeRefCounted<MockServiceB>());
  }
};
```

## Lifetimes of promises and objects bound to them

Promises are refcounted and objects bound to them will be destructed along with them. When a promise is resolved or rejected, depending on whether the bound object is move only or not, it's either copied or moved down the chain of promises (NB for move only types multiple Thens are not allowed). This means that it doesn't matter if something retains a reference to the original promise, the bound object will either get passed to a resolve or reject handler (who now owns it) or it will get deleted when the very last promise in the chain gets deleted.

A simple example:

```
{
  Promise<std::unique_ptr<Foo>> p = my_task_runner
      ->PostTask(FROM_HERE, BindOnce([](){ return std::make_unique<Foo>(); }));

  RunLoop().RunUntilIdle();

  // If |my_task_runner| runs tasks on this sequence then |p| and |Foo| will be
  // destructed at here at the end of this scope.  If |my_task_runner| posts tasks
  // on a different thread then destruction could happen on either this thread or the
  // other thread depending on where the refcount to |p| goes to zero.
}
```

If there was a ThenOn() clause, we know for sure where Foo gets deleted:

```
Promise<std::unique_ptr<Foo>> p = ThreadTaskRunnerHandle::Get()
    ->PostTask(FROM_HERE, BindOnce([](){ return std::make_unique<Foo>(); }));

p.ThenOn(my_task_runner,
         BindOnce([](std::unique_ptr<Foo> result) {
           // |Foo| gets destructed here on |my_task_runner|.
         });
```

Things get more complicated for unhandled rejections, consider:

```
Promise<void, std::unique_ptr<Err>> p1 = task_runner1
    ->PostTask(FROM_HERE,
               BindOnce([]() -> PromiseResult<void, std::unique_ptr<Err>> {
                 return std::make_unique<Err>();
               }));

Promise<void, std::unique_ptr<Err>> p2 =
    p1.ThenOn(task_runner2, FROM_HERE, BindOnce([]() {
                 ...
               }));
```

```
Promise<void, std::unique_ptr<Err>> p3 =
    p2.ThenOn(task_runner3, FROM_HERE, BindOnce([]() {
                ...
              }));
```

When |p1| rejects |p2| also rejects which causes |p3| to reject. At each stage ownership of Err is passed down the chain and Err will eventually get deleted on |task_runner3|.

Similar considerations apply to resolved promises and void callbacks, e.g:

```
Promise<std::unique_ptr<Foo>> p1 = task_runner1
    ->PostTask(FROM_HERE,
              BindOnce([]() {
                return std::make_unique<Foo>();
              }));

p2 = p1.ThenOn(task_runner2, FROM_HERE, BindOnce([]() {
                ...
              }));

p3 = p1.ThenOn(task_runner3, FROM_HERE, BindOnce([]() {
                ...
              }));
```

Foo will be deleted after |p2| and |p3| have run.  Since |p2| and |p3| are posted to different sequences, we can't be sure which sequence Foo will be deleted on.  NB it's illegal for p2 & p3 to both take ownership of Foo and the library will DCHECK if you try to do this.

## Benchmarks & Impact on code size

The prototype increases the size of an Android chrome APK by 28339 bytes.  Care has been taken with judicious use of NOINLINE to avoid bloat due to unwanted inlining.  It's not expected that users will typically have to use NOINLINE.

On linux with an ordinary release x64 build (no LTO which should reduce it) the binary delta for a new return type with this PostTaskAndReply replacement is 1680 bytes as measured by the code snippet below.  This delta is due to the various templates that implement promises, which have to handle the new MyFoo type.

```
#if 0
struct MyFoo {
  int a;
  int b;
};

MyFoo MyTask1() {
  return {10, 20};
}

void MyTask2(MyFoo v) {}
#else
void MyTask1() {}
void MyTask2() {}
#endif

base::ThreadTaskRunnerHandle::Get()
      ->PostTask(FROM_HERE, base::BindOnce(&MyTask1))
      .ThenHere(FROM_HERE, base::BindOnce(&MyTask2));
```

Returning a promise handle from PostTask does have some overhead which we can measure using the TaskQueueManagerPerftest microbenchmark.  Promises made no significant difference for the performance of immediate task posting.

Perf trybot runs:

| Benchmark | Result summary |
|---|---|
| thread_times.key_mobile_sites_smooth | No difference. |
| loading.mobile | No difference. |
| memory.top_10_mobile | No difference. |
| system_health.memory_mobile | No difference. |
| system_health.common_mobile | |
| power.typical_10_mobile | No difference. |
| start_with_url.cold.startup_pages | No difference. |
| system_health.webview_startup | |

**Memory allocation overhead**

Oct 25th - updated perf results:

System_health.memory_desktop some of these metrics are super noisy but it looks like we're increasing the size of malloc'ed memory by 0.1% which isn't overly concerning.

loading.mobile again there's a bunch of noise on these metrics, and I interpret this as meaing there is essentially no change.

Manual instrumentation:

I checked a simple PostTaskAndReply vs it's Promise based alternative. PostTaskAndReply resulted in 5 memory allocations. The promise alternative resulted in 7 memory allocations, the extra two being for AbstractPromise and AbstractPromise::Graph.

I also checked PostTask, the regular version has 2 allocations to do this, but the promise version had 3. The difference being 176 bytes for AbstractPromise.

```
class PostTaskTest : public testing::Test {
 public:
  test::ScopedTaskEnvironment scoped_task_environment_;
  RunLoop run_loop_;
};

TEST_F(PostTaskTest, PostTask) {
  __dbg = true;
  printf(">>>>> START <<<<<\n");
  ThreadTaskRunnerHandle::Get()
      ->PostTask(FROM_HERE, BindOnce([]() { printf("A\n"); }));
  run_loop_.RunUntilIdle();
  printf(">>>>> END <<<<<\n");
  __dbg = false;
}
```

## Implementation Order

1. After consulting with clang experts it has been decided we don't need to wait for C++ 17.
2. We will land C++14 support for base::unique_any (done) and base::variant.
3. Then we will start landing AbstractPromise which is the non-templatized ref-counted promise implementation. Each AbstractPromise can be thought of as a node in the graph of unresolved promises and each AbstractPromise contains links to pre-requisite

promises and to promises that are dependant on it.  The code is designed to be thread safe taking advantage of the PostTask locks and using atomics.  We will add extensive tests to make sure this core class works as expected.

4. We will land AbstractPromiseExecutor which does much of the heavy lifting template magic.
5. We will land the main Promise class, mostly to get the API right.  It will compile but will be mostly non-functional without PostTask support.
6. We will change PostTask to return a promise and base::PendingTask to contain a promise and land the bulk of the Promise tests. This will be a big but mostly mechanical patch
7. After that we will land Promises::All
8. (maybe) If needed we will land Promises::Race

## Options we decided to reject

- We thought about allowing multiple rejection types in a Then() chain, but we decided that was likely to lead to the context of the rejection reason becoming hard to understand. Particularly if types such as int are used for the reject reason. Instead when changing reject reason you will need to catch and rethrow using the new type. E.g. we won't support this:

```
Promise<int, bool> p;
p.ThenHere(FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
  .ThenHere(FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
  .ThenHere(FROM_HERE,
        base::BindOnce([](int result) -> base::PromiseResult<int, std::string> {
          // Note because we're changing the promise signature mid chain, the error
          // handler below needs to accept either a bool or a std::string.
          return std::string("Oh no!");
        }))
  .ThenHere(FROM_HERE,
        base::BindOnce([](int result) {
          // Handle result.
        }),
        base::BindOnce([](const Variant<bool, std::string>& error) {
          // Handle error.
        }));
```

  But we will support this:

```
Promise<int, bool> p;
p.ThenHere(FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
  .ThenHere(FROM_HERE, base::BindOnce([](int result) { return result + 1; }))
```

```
.ThenHere(FROM_HERE,
      base::BindOnce([](int result) -> base::PromiseResult<int, std::string> {
        if (result < 0) std::string("Oh no!"); // Reject
        return result; // Resolve
      },
      base::BindOnce([](bool err) -> base::PromiseResult<int, std::string> {
        // Convert the reject reason to the new type.
        return err ? std::string("Reason 1") : std::string("Reason 2");
      })
.ThenHere(FROM_HERE,
      base::BindOnce([](int result) {
        // Handle result.
      }),
      base::BindOnce([](const std::string& error) {
        // Handle error.
      }));
```

- Then & Catch implicitly running on the current sequence. The idea was unless otherwise specified Then and Catch would run on the same sequence as the previous promise task. The problem with that is developers are used to the semantics of PostTaskAndReply where the reply runs on the current sequence. We chose instead to be explicit about which sequence things are running on.
- Setting void as the default rejection type. We felt it was better if the type documented whether or not a promise could reject.
- Cancellation in V1. Getting the semantics right has proved to be hard because it's not obvious what cancelling a promise in the middle of a chain should mean. Obviously dependant promises would get canceled, but what about prerequisite promises?

## Future work

- It would be great if Chrome tracing could know more about promises - perhaps being able to visualize the graph of promises, and graphically highlighting the various operations: resolve, reject, race, all.
- Promise chains should have traits to enable the scheduler to prioritize them.
- Building support for coroutines to allow C++ code to await the result of a promise. The JS experience suggests this will greatly improve readability.

  The coroutines ts dovetails nicely with promises, allowing us to co_await promises which greatly simplifies async code. E.g. you can write functions like this. Proof of concept patch.

  ```
  Promise<void> MyTask() {
           ThingA a = co_await GetThingAPromise();
  ```

```
    ...

    Variant<Resolved<void>, Rejected<net::Error>> result =
       co_await GetPromiseThatMightReject();

    if (result.index() == 1) {
      // Handle rejection
    }

    ThingB b = co_await GetThingBPromise();

    ...
  }
```

- Utility methods to bridge from and to Java promises.
- Replace usages of PostTaskAndReply with promises.