C++ Promise/Future/Coroutine in Chromium

2018-01-18

Discussion note for C++ Promise and Future implementation.

Glossary

- Promise & Future:
  - In tzik's version, Future<X> represents a value that will be resolved to a X in the future, and Promise<X> represents a promise for the owner to give a X at some point. An associated pair of Promise and Future is a thread-safe oneshot channel from Promise to Future that notifies a promise resolution or rejection with a value.
  - In alexclarke's version, Promise<X> represents a value that will be resolved to a X, and PromiseResolver<X> is the other end to send a value to the corresponding Promise.

Prototype implementations:

- tzik: https://crrev.com/c/911230
- alexclarke: https://crrev.com/c/843976, https://crrev.com/c/858137

Comparison of the proposals:

	tzik	alexclarke	
Lifetime	Futures and Promises are move-only. Any of Resolve(), Reject(), Then(), and Catch() consumes the Future or Promise. Future passes the value by move, so that it can handle ownership correctly.	Promise is refcounted. This lets a single promise trigger multiple Thens.	**To be discussed**
Threading	Uses atomics for thread safety. Can be used from any thread without locks.	Not thread safe (yet).	**To be discussed**
Promise Rejection	Has typed rejection support. Can handle multiple rejection types.	Has rejection including support for multiple reject types in a promise chain.	
Promise Cancellation	Not supported.	Supported	**To be discussed.**
Finally	Supported	Supported	
Resolving a promise with a promise/future	Supported.	Supported	
Then & Catch can be	Supported	Supported	

chained[1]			
Multiple arguments support	Supported. Then() tries to apply the value directly, and if it fails, applies the value with unpacking as a tuple.	std::tuple can be unpacked and applied when running a Then or a Catch callback.	
variant<> implementation	Add std::variant<> clone	std::variant<> clone and custom AbstractVariant	**To be discussed**
PostTask support		Preliminary support	

Synopsis of tzik's proposal:

```
namespace base {

// Promise sends a resolution or rejection to the corresponding Future.
// The user may call either Resolve() or Reject() once, they can be called on any thread.
template <typename ResolveType, typename... RejectTypes>
class Promise {
 public:
  // Move-only type.
  Promise() = default;
  Promise(const Promise&) = delete;
  Promise(Promise&&) = default;
  Promise& operator=(const Promise&) = delete;
  Promise& operator=(Promise&&) = default;

  // Returns true if Resolve() or Reject() invocation is valid on this instance.
  // Returns false on the default constructed instance, moved instance,
  // and consumed instance by Resolve() or Reject() invocation.
  explicit operator bool() const;

  // Resolves the promise and notify associated Future of the resolution.
  template <typename... Args>
  void Resolve(const Location& from_here, Args&&...) &&;

  // Rejects the promise and notify associated Futere of the rejection.
  template <typename T, typename... Args>
  void Reject(const Location& from_here, Args&&...) &&;
};

// Future receives the resolution or rejection from the corresponding Promise.
// The user may call either My proposed solution to this is to support callbacks with and with out
references. If a reference isn't used then std::move() should be used.Then() or Catch() once, they can be
called on any thread.
template <typename ResolveType, typename... RejectTypes>
class Future {
 public:
  // Creates Future and associates it to |promise|.
  explicit Future(Promise<ResolveType, RejectTypes...>* promise);

  // Move-only type.
```

[1] E.g.

```
Promise<int, std::string> p;
p.Then(FROM_HERE, BindOnce([](int result) { return result + 1; }))
.Then(FROM_HERE, BindOnce([](int result) { return result + 1; }))
.Then(FROM_HERE, BindOnce([](int result) -> PromiseResult<int, std::string> {
    return std::string("Oh no!");
}))
.Then(FROM_HERE, BindOnce([](int result) {
        // We shouldn't get here, the promise was rejected!
      }),
      BindOnce([](const std::string& err) {
        EXPECT_EQ("Oh no!", err);
      }));
```

```
  Future() = default;
  Future(const Future&) = delete;
  Future(Future&&) = default;
  Future& operator=(const Future&) = delete;
  Future& operator=(Future&&) = default;

  // Returns true if Then() or Catch() invocation is valid on this instance.
  // Returns false on the default constructed instance, moved instance,
  // and consumed instance by Then() or Catch() invocation.
  explicit operator bool() const;

  // Then() sets a resolution handler, that takes the resolution value from
  // the corresponding Promise.
  // The resulting value of the handler is wrapped and passed as a new Future,
  // and returned from Then().
  template <typename R>
  Future<R, RejectTypes...> Then(
      const Location& from_here,
      scoped_refptr<TaskRunner> task_runner,  // optional.
      OnceCallback<R(ResolveType)> handler) &&;

  // A variant of Then() whose handler returns a Future for chaining the
  // asynchronous operations.
  // Then() flattens the resulting Future. The rejection types are merged
  // and deduped.
  template <typename R, typename... NewRejectTypes>
  Future<R, NewRejectTypes..., RejectTypes...> Then(
      const Location& from_here,
      scoped_refptr<TaskRunner> task_runner,  // optional
      OnceCallback<Future<R, NewRejectTypes...>(ResolveType)>) &&;

  // Catch() sets a rejection handler, that takes the rejection value from
  // the corresponding Promise.
  template <typename R>
  Future<ResolveType, /* Unhandled rejections */> Catch(
      const Location& from_here,
      scoped_refptr<TaskRunner> task_runner,  // optional
      OnceCallback<R(/* one of RejectTypes here */)>) &&;

  // A variant of Catch() whose handler returns a Future for chaining.
  template <typename R, typename... NewRejectTs>
  Future<ResolveType, NewRejectTypes..., RejectTypes...> Catch(
      const Location& from_here,
      scoped_refptr<TaskRunner> task_runner,
      OnceCallback<Future<R, NewRejectTypes...>(/* one of RejectType */)>) &&;
};

// Combines given Futures, and returns a Future that is resolved when
// all Futures are resolved.
// RejectT of them needs to have the same type.
template <typename... ResolveTs, typename RejectT>
Future<std::tuple<ResolveTs...>, RejectT> Combine(Future<ResolveTs, RejectT>...);

// Returns a Future that is resolved when all given Futures are resolved.
// RejectT of them needs to have the same type.
template <typename... ResolveT, typename RejectT>
Future<std::vector<ResolveT>, RejectT> All(std::vector<Future<ResolveT, RejectT>>);

template <typename... ResolveTs, typename RejectT>
Future<ResolveT, RejectT> Any(std::vector<Future<ResolveT, RejectT>>);

}  // namespace base
```

Alex's one:

```
// By default |RejectType| is void unless |ResolveType::DefaultRejectType|
// exists, in which case that will be used.
template <typename ResolveType,
          typename RejectType =
              typename internal::PromiseTraits<ResolveType>::DefaultRejectType>
class Promise {
 public:
```

```
  // Constructs an unresolved promise for use by a ManualPromiseResolver<>.
  Promise();

  // Helpers for computing Then/Catch promise return and reject types based on
  // the callback signature.
  template <typename CB>
  using ThenResolve = typename internal::ComputeThenPromise<
      typename internal::CallbackTypeHelper<CB>::ReturnT,
      RejectType>::ThenResolve;

  template <typename CB>
  using ThenReject = typename internal::ComputeThenPromise<
      typename internal::CallbackTypeHelper<CB>::ReturnT,
      RejectType>::ThenReject;

  template <typename CB>
  using CallbackReturnType = typename internal::CallbackTypeHelper<CB>::ReturnT;

  template <typename CB>
  using CallbackArgumentType = typename internal::CallbackTypeHelper<CB>::ArgT;


  // A task to execute |on_reject| is posted on the current
  // SequencedTaskRunnerHandle as soon as the this promise is rejected. A
  // Promise<> for the return value of |on_reject| is returned. The following
  // callback return types have special meanings:
  // 1. PromiseResult<Resolve, Reject> lets the callback resolve, reject or
  //    curry a Promise<Resolve, Reject>
  // 2. Promise<Resolve, Reject> where the result is a curried promise.
  template <typename RejectCb>
  Promise<ThenResolve<RejectCb>, ThenReject<RejectCb>>
  Catch(const Location& from_here, OnceCallback<RejectCb> on_reject);

  // As soon as the this promise is resolved, a task to execute |on_resolve|
  // is posted on the current SequencedTaskRunnerHandle. A Promise<> for the
  // return value of |on_resolve| is returned. The following callback return
  // types have special meanings:
  // 1. PromiseResult<Resolve, Reject> lets the callback resolve, reject or
  //    curry a Promise<Resolve, Reject>
  // 2. Promise<Resolve, Reject> where the result is a curried promise.
  template <typename ResolveCb>
  Promise<ThenResolve<ResolveCb>, ThenReject<ResolveCb>>
  Then(const Location& from_here, OnceCallback<ResolveCb> on_resolve);

  // As soon as the this promise is resolved or rejected, a task to execute
  // |on_resolve| or |on_reject| is posted on the current
  // SequencedTaskRunnerHandle. A Promise<> for the return value of
  // |on_resolve| or |on_reject| is returned. The following callback return
  // types have special meanings:
  // 1. PromiseResult<Resolve, Reject> lets the callback resolve, reject or
  //    curry a Promise<Resolve, Reject>
  // 2. Promise<Resolve, Reject> where the result is a curried promise.
  template <typename ResolveCb, typename RejectCb>
  Promise<ThenResolve<ResolveCb>, ThenReject<ResolveCb>>
  Then(const Location& from_here,
       OnceCallback<ResolveCb> on_resolve,
       OnceCallback<RejectCb> on_reject);

  // A task to execute |finally_callback| is posted after the parent promise is
  // resolved or rejected and all Thens have executed.
  void Finally(const Location& from_here, OnceClosure finally_callback);

  // Cancels this promise and others that depend on it, with special handling of
  // promise::Race() to ensure that is only canceled if all prerequisites are
  // canceled.
  void Cancel();
};

// Used for manually resolving and rejecting a Promise. This is for
// compatibility with old code and will eventually be removed.
template <typename ResolveType,
          typename RejectType =
              typename internal::PromiseTraits<ResolveType>::DefaultRejectType>
```

```
class ManualPromiseResolver {
 public:
  ManualPromiseResolver() {}

  template <typename... Ts>
  void Resolve(Ts... t);

  template <typename... Ts>
  void Reject(Ts... t);

  typename internal::PromiseCallbackHelper<ResolveType>::Callback
  GetResolveCallback();

  typename internal::PromiseCallbackHelper<RejectType>::Callback
  GetRejectCallback();


  Promise<ResolveType, RejectType> promise() const { return promise_; }

 private:
  Promise<ResolveType, RejectType> promise_;
};


// One of the supported promise result types. This allows a promise to
// dynamically reject or resolve (potentially with another promise).
template <typename ResolveType,
          typename RejectType =
              typename internal::PromiseTraits<ResolveType>::DefaultRejectType>
class PromiseResult {
 public:
  // No argument constructor, can either resolve or reject depending on the
  // template arguments.
  PromiseResult(void);

  // Resolve overrides
  PromiseResult(const Resolved<ResolveType>& resolved);
  PromiseResult(Resolved<ResolveType>&& resolved);

  PromiseResult(const ResolveType& resolved);
  PromiseResult(NonVoidResolveType&&) {}

  // Resolve with Curried promise
  PromiseResult(Promise<ResolveType, RejectType> curried_promise);

  // Reject overrides
  PromiseResult(const Rejected<RejectType>& rejected);
  PromiseResult(Rejected<RejectType>&& rejected);

  PromiseResult(const RejectType& rejected);
  PromiseResult(RejectType&& rejected);
};

class Promises {
 public:
  template <typename... Promises>
  struct VarArgIsPromise;

  /*
   * Accepts either a container of Promise<Resolve, Reject> and returns a
   * Promise<std::vector<Resolve>, Reject> or it accepts a container of
   * Variant<Promise<Resolve, Reject>...> and returns a
   * Promise<std::vector<Variant<Resolve...>>, Variant<Reject...>> which is
   * resolved when all input promises are resolved or any are rejected.
   */
  template <typename Container>
  static typename internal::
      AllContainerHelper<Container, typename Container::value_type>::PromiseType
      All(const Container& promises);

  /*
   * Accepts one or more promises and returns a
   * Promise<std::tuple<Resolve> ...>, Variant<Reject...>> which is resolved
```

```
   * when all promises resolved or rejects with the RejectValue of the first
   * promise to do reject.
   */
  template <typename... Resolve, typename... Reject>
  static Promise<std::tuple<internal::ToNonVoidT<Resolve>...>,
                 typename internal::UnionOfVarArgTypes<Reject...>::type>
  All(Promise<Resolve, Reject>... promises);

  /*
   * Accepts either a container of Promise<Resolve, Reject> and returns a
   * Promise<Resolve, Reject> or it accepts a container of
   * Variant<Promise<Resolve, Reject>...> and returns a
   * Promise<Variant<Resolve...>, Variant<Reject...>> which is resolved when any
   * input promise is resolved or rejected.
   */
  template <typename Container>
  static typename internal::RaceContainerHelper<
      Container,
      typename Container::value_type>::PromiseType
  Race(const Container& promises);

  /*
   * Accepts one or more promises and returns a
   * Promise<Variant<distinct non-void promise types>> which is resolved
   * when any input promise is resolved or rejected.
   */
  template <typename... Resolve, typename... Reject>
  static Promise<typename internal::UnionOfVarArgTypes<Resolve...>::type,
                 typename internal::UnionOfVarArgTypes<Reject...>::type>
  Race(Promise<Resolve, Reject>... promises);

  // Returns a rejected promise.
  template <typename ResolveType, typename RejectType>
  static Promise<ResolveType, RejectType> Reject(RejectType&& rejected);

  // Returns a rejected promise.
  template <typename ResolveType, typename>
  static Promise<ResolveType, void> Reject();

  // Returns a resolved promise.
  template <typename ResolveType,
            typename RejectType = typename internal::PromiseTraits<
                ResolveType>::DefaultRejectType>
  static Promise<ResolveType, RejectType> Resolve(ResolveType&& resolved);

  template <typename RejectType =
                typename internal::PromiseTraits<void>::DefaultRejectType>
  static Promise<void, RejectType> Resolve();
};
```

Use case:

```
// Returning result of Asynchronous operations with Promise/Future.
class FooService {
  public:
    base::Future<int, Error> Foo() {
      return base::MakeResolved(42);
    }
};

// Handling result.
service->Foo().Then(BindOnce([](int x) {
  LOG(INFO) << x;
  return x / 2;
})).Then(BindOnce([](int y) {
  LOG(INFO) << y;
})).Catch(BindOnce([](Error e) {
  // Error handling.
}));
```

```
// Promise/Future support of PostTask
PostTask([]() {
  return "Hello, world!";
}).Then(BindOnce([](const char* msg) {
  LOG(INFO) << msg;
}));
```

PostTask

gab@ Suggested we change the return type of PostTask to Promise to allow:

```
Promise<int> p = task_runner->PostTask(FROM_HERE, base::Bind([](){ return 1; }));
p.Then(FROM_HERE, base::Bind([](int value){ ... }));
...
p.Cancel();
```

Possible implementation notes:

- Changing base::PendingTask to contain a reference to the pending promise, so the TaskScheduler can check if the promise has been cancelled or not.
- Something (perhaps initially a template on TaskRunner) needs to post a wrapped task where the return value is fed into the promise machinery.

Non-void return value support of blink::CrossThreadBind

C++ Coroutines TS:
https://isocpp.org/files/papers/N4663.pdf
C++ Coroutines TS is available on clang with -fcoroutine-ts option. This probably misses C++20, but I assume it's worth making our Future/Promise to ready to support Coroutines. Both of the proposal seems to be able to support Coroutines.
After Coroutines get ready to use, we want to adopt it to our Promise/Future, so that we can write async operation as:

```
Future<int> Foo() { co_return 6; }
Future<int> Bar() { co_return 7; }

Future<double> Baz() {
  int x = co_await Foo();
  int y = co_await Bar();
  co_return x * y;
}
```

instead of

```
Future<double> Baz() {
  All(Foo(), Bar()).Then(BindOnce([](int x, int y) {
    return x * y;
  }));
}
```

- How should we handle rejections without exceptions?
- Coroutine TS moves local variables to a heap-allocated object, and Oilpan will fail to find these pointers on the stack scanning, that causes an over collection.
    - We may have to use blink::Persistent<> even on stack variables in the coroutine context, and check that with a clang plugin.

Discussion and next step

- Thread-safe promise or the combo of same thread & cross thread promise?
- Move-only or ref-counted?
    - We should make it move-only if we make it thread-safe. Otherwise, ref-counted works.
- Cancellation?
- Next step: post this to cxx@chromium.org for more feedback.

Discussion note: 2018-01-26

- https://docs.google.com/document/d/1Wg7cAY8aiZTmqBfkx3_ZIX6vkowCLJEofnUiXpRRddc/edit#bookmark=id.5qsjljd0mevf