# TaskTraits for sequence scheduling in //content

**PUBLIC DOCUMENT**

Authors: eseckler@
Last Updated: 2018-07-10
Tracking bug: https://crbug.com/867421
Patches: [1], [2]

## Overview

To prioritize tasks on the browser's IO and UI thread dynamically (see UI thread scheduling), we need to annotate tasks with attributes and tokens that //content's SequenceManagers and their schedulers can later use to make scheduling decisions, such as existing base::TaskTraits, the type of task (e.g. net, compositing, ..), the id of the frame a task is associated with, and other attributes (e.g. whether the task is load or startup blocking).

For this purpose, we propose to extend the TaskScheduler's TaskTraits concept to support additional "extension" traits and tokens provided from outside //base. Traits specific to //content will include the attributes and tokens mentioned above, as well as the browser thread that the task should run on.

The API to post tasks will remain in base's post_task.h but TaskScheduler will allow external TaskExecutors to be registered. Tasks posted with TaskTraits that have the extended attributes will be forwarded to the appropriate TaskExecutor (template magic).

This base API to post tasks will replace the existing PostTask() methods on browser_thread.h, so that we have a single unified way to post tasks. Blink may eventually also switch to this API.

## API proposal

### Goal: Examples for posting tasks

```
// Post a task to the TaskManager's thread pool (no change from today's code):
base::PostTaskWithTraits(FROM_HERE, {
    base::TaskPriority::BACKGROUND,
    base::MayBlock()}, base::Bind(...));

// Post a network task associated with a specific frame to the browser's UI thread.
base::PostTaskWithTraits(FROM_HERE, {
    content::BrowserThread::UI,
    content::TaskType::kNetwork,
    content::FrameToken(render_frame_host->GetDevToolsFrameToken()),
    content::BlocksLoad()}, base::Bind(...));
```

Tasks that are posted with the same traits will be scheduled in the order they were posted. If the traits imply a single sequence (e.g. because a BrowserThread was specified in them), this means that they will execute sequentially. If the traits imply a WorkerPool, the exact execution order is not guaranteed.

We will also support these traits in CreateSequenced/SingleThreadTaskRunnerWithTraits(), with the same sequencing behavior (i.e. two TaskRunners with the same traits will schedule tasks sequentially if the traits imply a single sequence).

## API / Implementation sketch

Embedders can attach extension traits to a TaskTraits object in a way that is opaque to base. These extension traits can then be specified along the base traits when constructing the TaskTraits object using the current braces syntax (constexpr TaskTraits t = {MyExtensionTrait()}). They are then stored and propagated with the TaskTraits object.

To support constexpr-compatible construction, extension traits are stored in a fixed-size byte array in the TaskTraits object and serialized into and parsed from this storage by an embedder-provided extension class and MakeTaskTraitsExtension() template function.

### Modifications to TaskTraits / posting tasks (base)

```
namespace base {

// Stores extension traits opaquely inside a fixed-size data array. We store
// this data directly (rather than in a separate object on the heap) to support
// constexpr-compatible TaskTraits construction.
struct BASE_EXPORT TaskTraitsExtensionStorage {
  static constexpr size_t kStorageSize = 8;  // bytes
  static constexpr uint8_t kInvalidExtensionId = 0;

  inline constexpr TaskTraitsExtensionStorage() {}
  inline constexpr TaskTraitsExtensionStorage(
      uint8_t extension_id,
      std::array<uint8_t, kStorageSize>&& data)
      : extension_id(extension_id), data(std::move(data)) {}

  inline constexpr TaskTraitsExtensionStorage(
      const TaskTraitsExtensionStorage& other) = default;
  inline TaskTraitsExtensionStorage& operator=(
      const TaskTraitsExtensionStorage& other) = default;

  // Identifies the type of extension. Value 0 denotes an invalid ID; values 1
  // to 9 are reserved for testing purposes by base. The embedder is responsible
  // for assigning the remaining values (10 to 255) uniquely.
  uint8_t extension_id;

  // Serialized extension data.
  std::array<uint8_t, kStorageSize> data;
};
```

```
// Default implementation of MakeTaskTraitsExtension template function, which
// doesn't accept any traits and does not create an extension.
template <class Unused = void>
void MakeTaskTraitsExtension(TaskTraitsExtensionStorage& storage) {}

// Forwards those arguments from |args| that are indicated by the index_sequence
// to a MakeTaskTraitsExtension() template function, which is provided by the
// embedder in an unknown namespace; its resolution relies on argument-dependent
// lookup.
template <class... ArgTypes, std::size_t... Is>
constexpr void MakeTaskTraitsExtensionHelper(
    TaskTraitsExtensionStorage& storage,
    std::tuple<ArgTypes...> args,
    std::index_sequence<Is...>) {
  MakeTaskTraitsExtension(
      storage, std::get<Is>(std::forward<std::tuple<ArgTypes...>>(args))...);
}

class TaskTraits {
 private:
  struct ValidTrait {
    ValidTrait(TaskPriority) {}
    ValidTrait(TaskShutdownBehavior) {}
    ValidTrait(MayBlock) {}
    ValidTrait(WithBaseSyncPrimitives) {}
  };

 public:
  template <class... ArgTypes,
            class CheckArgumentsAreValidBaseTraits = std::enable_if_t<
                trait_helpers::AreValidTraits<ValidTrait, ArgTypes...>::value>>
  constexpr TaskTraits(ArgTypes... args) { ... }
  // Construct TaskTraits with extension traits.
  template <class... ArgTypes,
            class AvoidConstructorRedeclaration = void,
            class CheckArgsContainNonBaseTrait = std::enable_if_t<
                !trait_helpers::AreValidTraits<ValidTrait, ArgTypes...>::value>>
  constexpr TaskTraits(ArgTypes... args)
      // Select those arguments that are valid base TaskTraits and forward them
      // to the above constructor via a helper constructor.
      : TaskTraits(std::forward_as_tuple(args...),
                   trait_helpers::SelectIndices<
                       trait_helpers::ValidTraitTester<ValidTrait>::IsValid,
                       ArgTypes...>{}) {
    // Select all other arguments and try to create an extension with them.
    MakeTaskTraitsExtensionHelper(
        extension_, std::forward_as_tuple(args...),
        trait_helpers::SelectIndices<
            trait_helpers::ValidTraitTester<ValidTrait>::IsInvalid,
```

```
            ArgTypes...>{});
  }

  // Access the extension data by parsing it into the provided extension type.
  template <class TaskTraitsExtension>
  const TaskTraitsExtension GetExtension() const {
    DCHECK_EQ(TaskTraitsExtension::kExtensionId, extension_.extension_id);
    return TaskTraitsExtension::Parse(extension_);
  }

  uint8_t extension_id() const {
    return extension_.extension_id;
  }

 private:
  // Helper constructor which selects those arguments from |args| that are
  // indicated by the index_sequence and forwards them to the public
  // constructor.
  template <class... ArgTypes, std::size_t... Is>
  constexpr TaskTraits(std::tuple<ArgTypes...> args, std::index_sequence<Is...>)
      : TaskTraits(
            std::get<Is>(std::forward<std::tuple<ArgTypes...>>(args))...) {}

  TaskTraitsExtensionStorage extension_;

  ...
}

// A TaskExecutor allows posting tasks with custom traits and is registered into a global
// |g_task_executors_| set.
class TaskExecutor {
 public:
  // Returns |true| if the executor will handle tasks with the given traits.
  virtual bool WillHandleTaskWithTraits(const TaskTraits& traits) = 0;

  // Returns |false| if posting the task failed, e.g. during shutdown.
  virtual bool PostTaskWithTraits(...) = 0;
};
// TaskExecutors need to be registered before threads are created (PreCreateThreads()).
void RegisterTaskExecutor(TaskExecutor* task_executor);
void UnregisterTaskExecutorForTest(TaskExecutor* task_executor);

// Posting a task now checks for the type of the traits and either schedules it on the
// worker pool or asks the TaskExecutor to schedule it instead. Returns |false| on shutdown.
bool PostTaskWithTraits(const Location& from_here,
                        const TaskTraits& traits,
                        OnceClosure task) {
  if (traits.extension_id() != TaskTraitsExtensionStorage::kInvalidExtensionId) {
    // Give registered TaskExecutors a chance to handle execution of the task.
    for (auto& task_executor : g_task_executors_) {
```

```
      if (task_executor->WillHandleTaskWithTraits(traits)) {
        return task_executor->PostTaskWithTraits(from_here, traits, std::move(task));
      }
    }
  }

  // If no extension traits exist or no TaskExecutor wants to take it, schedule the task
  // into the appropriate worker pool instead.
  return g_task_scheduler_executor_->PostTaskWithTraits(from_here, traits, task);
}

} // namespace base
```

## Adding //content-specific TaskTraits

```
namespace content {

enum class TaskType { kNetwork, ... };
struct BlocksLoad {};
struct FrameToken {
  constexpr FrameToken(const base::UnguessableToken& devtools_token) :
      devtools_token(devtools_token) {}
  const base::UnguessableToken devtools_token;
}
class CONTENT_EXPORT ContentTaskTraitsExtension {
 public:
  static constexpr uint8_t kExtensionId = 10;

  struct ValidTrait {
    ValidTrait(BrowserThread::ID) {}
    ValidTrait(TaskType) {}
    ValidTrait(BlocksLoad) {}
    ValidTrait(FrameToken) {}
  };

  template <
      class... ArgTypes,
      class CheckArgumentsAreValid = std::enable_if_t<
          base::trait_helpers::AreValidTraits<ValidTrait, ArgTypes...>::value>>
  constexpr ContentTaskTraitsExtension(ArgTypes... args) { ... }

  constexpr base::TaskTraitsExtensionStorage Serialize() const {
    return {kExtensionId, {static_cast<uint8_t>(browser_thread_), ...}};
  }

  static const ContentTaskTraitsExtension Parse(
      const base::TaskTraitsExtensionStorage& extension) {
    return ContentTaskTraitsExtension(
```

```
        static_cast<BrowserThread::ID>(extension.data[0]), ...);
  }
  ...
};

template <class... ArgTypes,
          class = std::enable_if_t<base::trait_helpers::AreValidTraits<
              ContentTaskTraitsExtension::ValidTrait,
              ArgTypes...>::value>>
constexpr void MakeTaskTraitsExtension(
    base::TaskTraitsExtensionStorage& extension,
    ArgTypes&&... args) {
  extension =
      ContentTaskTraitsExtension(std::forward<ArgTypes>(args)...).Serialize();
}

// UI thread scheduler (or some proxy on its behalf) handles tasks destined for UI thread.
class UIScheduler : public base::TaskExecutor {
  ...

  bool WillHandleTaskWithTraits(const TaskTraits& traits) override {
    if (traits.extension_id() != ContentTaskTraitsExtension::kExtensionId)
      return false;

    auto content_traits = traits.GetExtension<ContentTaskTraitsExtension>();
    if (content_traits.browser_thread() != BrowserThread::UI)
      return false;

    return true;
  }

  void PostTaskWithTraits(const Location& from_here,
                          const TaskTraits& traits,
                          OnceClosure task) override {
    DCHECK_EQ((traits.extension_id(), ContentTaskTraitsExtension::kExtensionId);
    auto content_traits = traits.GetExtension<ContentTaskTraitsExtension>();
    DCHECK_EQ(content_traits.browser_thread(), BrowserThread::UI);
    // Determine correct queue and post the task.
    ...
  }
}

} // namespace content
```

## Alternative implementation approaches considered

### Alternative TaskTraits classes for base and embedders

Initially, we were considering separating the existing base TaskTraits from the TaskTraits used by embedders. That is, the current TaskTraits would become “TaskSchedulerTaskTraits”, and content would specify their own “ContentTaskTraits”, which may include a subset of base’s traits (or even extend TaskSchedulerTaskTraits). In this setup, the generic base::TaskTraits type would contain storage for any type of trait (think, std::any).

We discarded this idea in favor of extension traits because the latter would also allow us to specify extension traits for Tasks handled by the TaskScheduler. This way, we leave the door open to adding support for e.g. reprioritizing pending or future TaskScheduler tasks that were posted with specific extension traits.

### Indirect storage of extensions / specialized TaskTraits

Instead of a fixed-size storage for extensions (or specialized TaskTraits types as described in the section above), we could allocate extension traits on the heap and store a pointer into the TaskTraits. While this would simplify the storage and save space when no extensions are present, it prevents constexpr construction of TaskTraits (since memory allocation is not allowed). This turned out to have a not insignificant binary size impact (~35kB). We decided against this for that reason.