Proposal and High Level Design

Author: Justin Lulejian

Goal

Crbug

Propose a design that will do two things:

1. Simplify the logic for handling Service Worker “readiness” (to receive extension events) from its current complex form
2. Avoid requesting a worker to start when it’s already started (since this is unnecessarily done much of the time)

Solution

Use the State design pattern. In the state pattern there is an abstract State class, then N classes that inherit from the State to represent each state, and an interface class that is used to interact with each State transparently. In our solution we’ll call these:

- State: `ServiceWorkerState`
- States:
  - `ServiceWorkerReady`
  - `ServiceWorkerNotReady`
- State interface class: `ServiceWorkerInstance`

Proposal

<...> == existing code that is elided for conciseness

Notes:

- This design requires some `ServiceWorkerContextObserver` methods to be synchronously called.
- `ServiceWorkerTaskQueue` adding itself as an observer of `ServiceWorkerInstance` is not shown for conciseness.

High-level Design

```
// service_worker_task_queue.h

class ServiceWorkerInstance : ..., public ServiceWorkerStateObserver {
  <...>
};
```

```
// service_worker_task_queue.cc

void ServiceWorkerTaskQueue::MaybeDispatchTask(PendingTask task) {

  // Alternative: move AddPendingTask() existing code to here.

  if (worker_instance_.Running()) {
    DispatchTaskImmediately(task);
    return;
  }

  // Dispatch after start worker via observer method WorkerHasStarted().
  // WorkerState would be replaced by ServiceWorkerInstance, but `this` keeps
  // `pending_tasks_`.
  pending_tasks_.push_back(task);
  worker_instance_.StartWorker();
}

void ServiceWorkerTaskQueue::ShouldEnqueueTask() {
  return !worker_instance_.Running();
}

// Similar to, but replacing ServiceWorkerTaskQueue::AddPendingTask(...).
// Strikethrough would be code that would be deleted. Alternatively,
// fold existing code into MaybeDispatchTask().
void ServiceWorkerTaskQueue::AddPendingTaskAndMaybeDispatch(
    const LazyContextId& lazy_context_id,
    PendingTask task) {
  <...same as existing code...>
  const SequencedContextId context_id = {lazy_context_id.extension_id(),
                                         lazy_context_id.browser_context(),
                                         *activation_token};
  WorkerState* worker_state = GetWorkerState(context_id);
  DCHECK(worker_state);
  auto& tasks = worker_state->pending_tasks_;
  // worker_state->pending_tasks_ having tasks means the
  // worker has been requested to start and hasn't started yet. So
  // `tasks.empty()` `false` means the worker is starting. `tasks.empty()`
  // `true` means that we don't know if the worker is started so we'll try to
  // start it to ensure it'll be ready for the task. This efficiency relies on
  // the assumption that only this boolean controls whether we request the
  // worker to start below.
  bool needs_start_worker = tasks.empty();
  tasks.push_back(std::move(task));

  if (worker_state->registration_state_ != RegistrationState::kRegistered) {
```

```
    // If the worker hasn't finished registration, wait for it to complete.
    // DidRegisterServiceWorker will Start worker to run |task| later.
    return;
  }

  // Start worker if there aren't any tasks to dispatch to the worker (with
  // `context_id`) in progress. Otherwise, assume the presence of pending tasks
  // means we've started the worker and our start worker callback will run the
  // pending tasks for us later.
  if (needs_start_worker) {
    RunTasksAfterStartWorker(context_id);
  }

  MaybeDispatchTask(task);
}

void ServiceWorkerTaskQueue::DispatchTaskImmediately((const PendingTask& task) {
  // Essentially a duplicate of existing RunPendingTasksIfWorkerReady().
}

void ServiceWorkerTaskQueue::DispatchTasksImmediately((std::vector<PendingTask>
tasks) {
  for (const auto& task : tasks ) {
    DispatchTaskImmediately(task);
  }
}

// Observer method called when the worker has just become fully started/ready.
void ServiceWorkerTaskQueue::WorkerHasStarted() {
  if (!pending_tasks_.empty()) {
    DispatchTasks(pending_tasks_);
  }
  pending_tasks_.clear();
}

private:
  ServiceWorkerInstance worker_instance_;
```

```
// service_worker_state.h

// ServiceWorkerState

class ServiceWorkerState  {
 public:
```

```
 virtual bool Running();

 // Calls ServiceWorkerContext::StartWorkerForScope() and
 // RegisterServiceWorker()
 virtual bool StartWorker();

 private:
   // Not explicitly shown, but used to get info for
   // LazyContextTaskQueue::ContextInfo for task dispatch
   std::optional<WorkerId> worker_id_;
   // Prevents redundant attempts to start the worker.
   bool worker_starting_;
};

class ServiceWorkerState::ServiceWorkerNotRunning : public ServiceWorkerState  {
 public:
   // false
   bool Running() override;

   // Calls ServiceWorkerContext::StartWorkerForScope() and maybe
   // RegisterServiceWorker()
   void StartWorker() override;
};

class ServiceWorkerState::ServiceWorkerRunning: public ServiceWorkerState  {
 public:
   // true
   bool Running() override;

   // no-op
   void StartWorker() override;
};
```

ServiceWorkerInstance with ServiceWorkerState pattern:

```
// service_worker_instance.h

class ServiceWorkerInstance : public ServiceWorkerState {
 public:

   // state_.Running();
   bool Running();

   // state_.StartWorker();
   void StartWorker();
```

```
 private:
   // Worker state monitoring

   // Taken from ServiceWorkerTaskQueue.
   // After all below called swaps state_ to ServiceWorkerRunning
   // calls ServiceWorkerStateObservers::WorkerHasStarted().
   void DidStartWorkerContext();
   void DidStartWorkerForScope();

   // ServiceWorkerContextObserver:
   // might be redundant with DidStartWorkerForScope()
   void OnVersionStartedRunning();
   // state_ == ServiceWorkerNotRunning, needs synchronous call
   void OnVersionStoppedRunning();

   // RenderProcessHostObserver:
   void RenderProcessExited();
   void RenderProcessHostDestroyed();

   // Worker state and observers.

   // ServiceWorkerRunning or ServiceWorkerNotRunning
   ServiceWorkerState state_;
   base::ObserverList<ServiceWorkerStateObserver> observers_;
};
```

Detailed Design Options

Design with State Pattern

Design without State Pattern (this pattern was chosen for simplicity)

Design w/ State Pattern

ServiceWorkerInstance fleshed out more with the state pattern:

```
// service_worker_instance.cc

bool ServiceWorkerInstance::Running() {
  return state_.Running();
}

bool ServiceWorkerInstance::StartWorker() {
  CHECK(!Running());
  return state_.StartWorker();
}

void ServiceWorkerInstance::OnVersionStartedRunning() {
  worker_started_browser_ = true;
  CheckWorkerRunningAndMaybeNotifyObservers();
}

void ServiceWorkerInstance::OnVersionStoppedRunning() {
  worker_started_browser_ = false;
}

void ServiceWorkerInstance::DidStartWorkerContext() {
  worker_started_renderer_ = true;
  CheckWorkerRunningAndMaybeNotifyObservers();
}
void ServiceWorkerInstance::DidStartWorkerForScope() {
  worker_started_browser_ = true;
  CheckWorkerRunningAndMaybeNotifyObservers();
}

void ServiceWorkerInstance::CheckWorkerRunningAndMaybeNotifyObservers() {
  if (Running()) {
    worker_starting_ = false;
    for (auto& observer : observers_) {
      observer.WorkerHasStarted();
    }
  }
}


// service_worker_instance.h (in addition to above definition)

bool worker_started_renderer_;
bool worker_started_browser_;
```

```
bool worker_starting_;
base::ObserverList<ServiceWorkerStateObserver> observers_;

// service worker states impls .cc

class ServiceWorkerState::ServiceWorkerNotRunning : public ServiceWorkerState  {
 public:
   bool Running() { return false; };

   void StartWorker() {
     if (!worker_starting_) {
       service_worker_context->StartWorkerForScope(
         ..., /*info_callback=*/DidStartWorkerForScope, ....);
       worker_starting_ = true;
  }
   };
};

class ServiceWorkerState::ServiceWorkerRunning: public ServiceWorkerState  {
 public:
   // true
   bool Running() {return true; };

   void StartWorker() { // no-op };
};
```

Design w/out State Pattern

ServiceWorkerInstance as a single class without the state pattern:

```
// service_worker_instance.cc

bool ServiceWorkerInstance::Running() {
  return worker_started_browser_ && worker_started_renderer_;
}

bool ServiceWorkerInstance::StartWorker() {
  CHECK(!Running());
  if (!worker_starting_) {
    service_worker_context->StartWorkerForScope(
      ..., /*info_callback=*/DidStartWorkerForScope, ....);
    worker_starting_ = true;
  }
}

<...same as the state pattern version...>
```