THIS IS A DEAD DOCUMENT

Activity Traits

Tracking resource consumption in Chromium

Author	chrisha@chromium.org
Last Edited	14 Feb 2017
Status	Abandoned
Bug	TBD
Reviewers	skyostil@chromium.org nduca@chromium.org gab@chromium.org

Introduction

The Chromium browser is a complex piece of software that places significant demands on system resources. It's multi-process nature means that many processes compete with each other for the use of these resources, in addition to external processes running on the system. Various *coordinator* efforts (MemoryCoordinator, TaskScheduler, etc) attempt to explicitly track and dynamically *coordinate* sharing of resources across the Chromium process tree. These efforts have been largely independent and orthogonal, each focused on an individual resource (eg, memory, CPU, disk, network, etc). Oftentimes reducing use of one resource will incidentally cause an increase in consumption of another (for example, trading memory for CPU). These competing resource coordinators can therefore unintentionally cause stress to be placed on other resources. Similarly, each of these subsystems uses its own measurement, aggregation and prioritization logic. These can introduce different worldviews and cause unintentional priority inversions. The GlobalResourceCoordinator is an effort to unify these various systems into one cohesive whole that is able to make *active* decisions about *trade-offs* and *consistent* decisions about relative priorities of components (eg, subsystems, services, frames, tabs, etc).

The first step along this path is to ensure that measurement of resource consumption is performed in a consistent manner across all of Chrome. Activity Traits seeks to unify the various existing mechanisms for aggregating performance metrics, and attributing costs to components, tasks, systems, events, etc. It is intended to unify various similar subsystems, and serve as a common basis for aggregation of various performance metrics (memory, CPU, disk, network, power), as well as provide the context under which runtime scheduling and resource distribution decisions are made. Consider this component as providing the tempo and time signature for a symphony played by an orchestra. Only once a common clock has been established can different performers reason about the performance of others, and can the composer evaluate the performance of the whole.

Guiding Principles

The Activity Traits system will always be running so performance implications are critical. The following are the overall guiding principles for the design:

- No dependencies. This component needs to be visible to all code in the Chromium codebase. Any dependencies should be header only, allowing it to be reused by components that can't even take a dependency on base (Blink, V8, etc).
- Minimal CPU overhead. Annotation and aggregation should be very quick. Overhead should be < 1%, and not user perceptible.
- Minimal memory overhead. Data storage should be compact and dense for improved cache performance. Overhead should be less than 1% and should scale linearly with the overall memory use of a given process, rather than monotonically with time.
- Data representations should facilitate slicing and aggregation. This will allow the system to be used as an efficient data source by real-time dashboards that allow roll-ups and drill-downs.
- Self-describing and persistent. The data representation should facilitate long-term storage of historical aggregate data for use by non-realtime timelines (chrome://tracing, for example).
- Expressive. The system needs to be sufficiently expressive that the needs of the various mechanisms it aims to replace can be met with little effort or refactoring.

The above list is a generic superset of requirements for various systems that ActivityTraits aims to replace, or at least supplement. Individual systems and transition plans will be discussed later.

Design

Concept

This design borrows very heavily from the initial TaskTraits[1] proposal for what eventually became FrameBlamer.

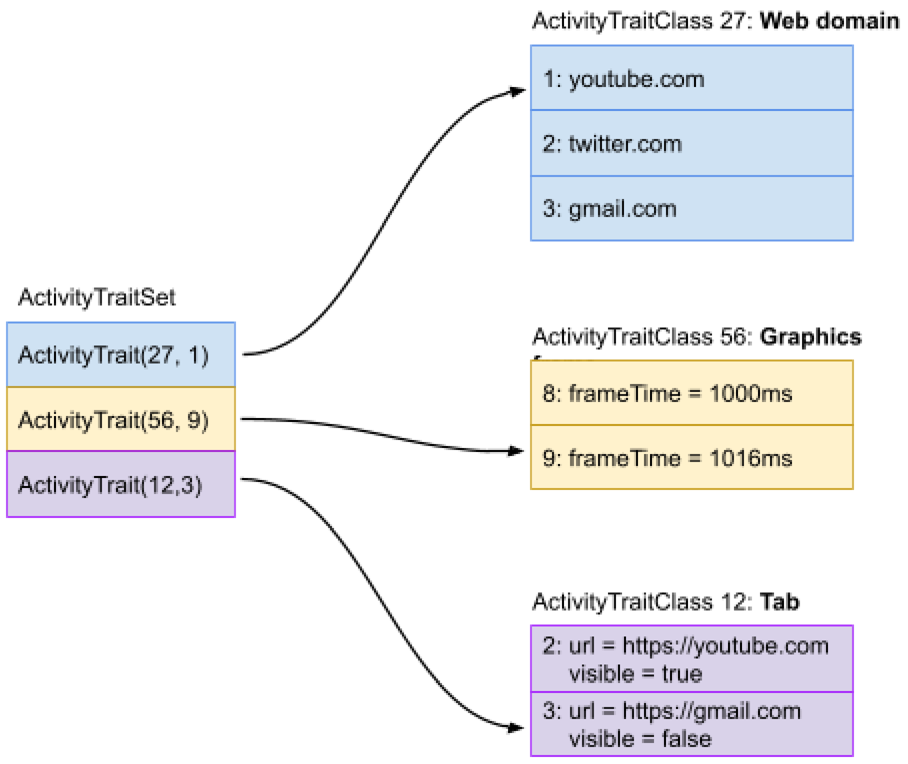
Activity Traits is about associating tasks in Chromium with a set of facts that provide information about each task's origin and other properties. A trait could be:

- A web domain (WebSecurityOrigin)
- A loaded resource (Javascript source file, HTML file, CSS file, image, etc)
- A Chromium subsystem or service (History, Sync, etc)
- A single graphics frame in a linear sequence of rendered frames
- One or a sequence of input events
- A V8 isolate
- A frame in a FrameTree
- A process
- A tab in the tab strip model
- A visible window on the user's desktop

Attribution makes it possible to directly measure and inspect why a resource is being consumed, and answer questions about both the where and the why of task origins. This data can be used by a offline tracing systems to provide web developers with visibility into their content, by real-time task managers to show where system resources are actively being consumed, and by coordinators to see where resources are being consumed and actively manage said consumption.

[1] A name that is overloaded, and now used for another meaning (see base::TaskTraits).

Architectural Overview


ActivityTraitClass 27: **Web domain**
1: youtube.com
2: twitter.com
3: gmail.com
ActivityTraitSet
ActivityTrait(27, 1)
ActivityTrait(56, 9)
ActivityTrait(12,3)
ActivityTraitClass 56: **Graphics**
8: frameTime = 1000ms
9: frameTime = 1016ms
ActivityTraitClass 12: **Tab**
2: url = https://youtube.com visible = true
3: url = https://gmail.com visible = false

An *ActivityTrait* is an opaque lightweight handle which identifies both a *trait class* as well as a specific *trait object* inside that class. As an example, a web domain would be a trait class (*ActivityTraitClass*) and the google.com domain would be a trait object within that class. Note that traits are specifically designed not to contain memory pointer data to make it feasible to pass them over IPC safely and efficiently.

```
// It is anticipated that there be a max of O(100s) of trait classes, but not O(1000s).
// These should be fixed and not reused. If a class is removed, it's ID should be
// deprecated. New classes should be added strictly at the end. This ensures
// forward and backwards compatibility of offline trace files.
enum class ActivityTraitClassId : uint32_t {
  // A static set of all known traits classes.
  RAIL_MODE = 0,
  WEB_DOMAIN = 1,
  GRAPHICS_FRAME = 2,
  BROWSER_TAB = 3,
  BROWSER_WINDOW = 4,
  BROWSER_PROFILE = 5,
  WEB_SECURITY_ORIGIN = 6,
  DEPRECATED_CLASS_THAT_NO_LONGER_EXISTS = 7,
  // ...
  MAX,
  INVALID = 0xFFFFFFFF,
};
```

```
using ActivityTraitObjectId = uint32_t;
constexpr ActivityTraitObjectId ACTIVITY_TRAIT_OBJECT_INVALID = 0xFFFFFFFF;

class ActivityTrait {
  ActivityTrait(ActivityTraitClassId class_id, ActivityTraitObjectId object_id);

  ActivityTraitClassId class_id() const;
  ActivityTraitObjectId object_id() const;
};
```

An *ActivityTraitClass* maintains the set of active objects of a specific trait type. For example, the web domain trait class would contain all the currently active domain names. Each trait object can be of an arbitrary type. Each *ActivityTraitClass* contains an injected *Delegate* object that defines the concrete type as well as a mechanism for serializing the type for offline tracing, or building a human-friendly string representation for online inspection. The set of trait classes is itself fixed and known completely at link time. Each trait class can be of two fundamental types: *static* or *dynamic*. A *static* trait class is associated with a fixed and finite enumeration of objects that are all known at compile time (e.g. RAIL mode). A *dynamic* trait class is associated with a set of objects that can grow and shrink dynamically at runtime (e.g. web domain).

```
// A Delegate is expected to implement an interface analogous to the following:
class BrowserProfileActivityTraitClassDelegate {
 public:
  // The underlying object type represented by this trait class.
  using Type = Profile*;

  // Converts a given instance of an object to a string description of it. Used for
  // user inspection of activity traits.
  static std::string ToString(Profile*) const;

  // Converts a given instance of an object into a serialized version of it. This
  // needs to be sufficiently detailed for a process to resolve/construct identical
  // instances of an object from a remote process.
  static void Serialize(Profile*, std::vector<uint8_t>* buffer);
};

template <ActivityTraitClassId class_id,
          const char* class_name,
          bool objects_are_dynamic,
          typename DelegateType>
class ActivityTraitClass {
 public:
  using Delegate = DelegateType;

  static const TraitClassId id = class_id;
  static const char* name = class_name;
  static const bool is_dynamic = objects_are_dynamic;

  // Returns a global instance of this class.
  static TraitClass<class_id, class_name, Delegate>* GetInstance();

  ActivityTrait GetOrCreate(const Delegate::Type& object);
  void Remove(const Delegate::Type& object);
  const Delegate::Type* Resolve(ActivityTraitObjectId);
};
```

A set of *ActivityTrait* objects is collected together into an *ActivityTraitSet*.

```
// This class is not thread-safe.
// TODO(chrisha): Decide whether this is a stack of traits, or a bag of traits.
class ActivityTraitSet {
 public:
  // Adds a single trait to this set. If the ActivityTraitClassId is already present in this
  // trait set then replaces the associated object_id.
  void AddOrReplace(ActivityTrait activity_trait);

  // Removes a specific activity trait. Returns true if it was present, false otherwise.
  bool Remove(ActivityTrait activity_trait);

  // Removes the trait associated with the provided class_id. Returns true if it was
  // present, false otherwise.
  bool Remove(ActivityTraitClassId class_id);

  // Returns a hash of this trait set. Useful for quickly resolving collisions in a
  // collection of trait sets.
  uint32_t Hash() const;

  // Returns the object ID associated with the given class ID in this trait set.
  // Returns ACTIVITY_TRAIT_OBJECT_INVALID if the class is not present.
  ActivityTraitObjectId GetObjectId(ActivityTraitClassId class_id);

  // Allow iterating over the ActivityTraits. Underlying storage will likely be
  // a vector-backed map.
  using iterator = ...;
  using const_iterator = ...;
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
};
```

Aggregation of statistics collected with respect to an activity will roll up to an *ActivityAggregates* object. This will hold aggregates of top-level performance statistics that are being tracked such as "memory allocated under this context", "memory freed under this context", "CPU time consumed under this context", "time spent waiting under this context", etc. The details of this object will be discussed later.

An *ActivityContext* will associate a constant *ActivityTraitSet* with a mutable set of aggregate data. Attempting to create an *ActivityContext* with a trait-set that is identical to a context that already exists should simply return a pointer to the same context. The *ActivityContex* is the fundamental unit that will be attached to a particular task, task runner or thread.

```
class ActivityContext {
 public:
  const ActivityTraitSet& trait_set() const;
  ActivityAggregates* aggregates();

  // TODO(chrisha): A reference counting mechanism. This will allow the context to be kept
  // alive by detailed memory allocation annotations, for example. This also allows the
context
  // to be kept alive by a reference to it from a pending task, or a task runner.
  // RefCounted might not be sufficient for this.
};
```

ActivityContexts are managed by the *ActivityContextManager*. This is a singleton in a process that needs to maintain its own thread-safety as it can and will be accessed from all thread. These will eventually be aggregated back to a central service.

```
// Responsible for creating and cleaning up ActivityContexts.
class ActivityContextManager {
 public:
  // Gets the active context for the current thread. This can be nullptr.
  ActivityContext* GetActiveContext();

  // Gets the context that was active during the task that posted the currently executed
  // task. This can be nullptr.
  ActivityContext* GetParentContext();

 protected:
  friend base::TaskRunner;

  // Creates a context for the provided ActivityTraitSet. If a context already exists
  // returns that instead.
  ActivityContext* GetOrCreate(const ActivityTraitSet& activity_traits);

  // Allows task-runners to set the current context pair.
  void SetContexts(ActivityContext* parent, ActivityContext* active);

  // TODO(chrisha): Efficient mechanisms for cleaning up and merging all contexts that
  // contain a specific trait. This clean-up needs to occur when an object is removed from a
  // dynamic trait class. Any aggregate data needs to be merged to a context with a reduced
  // trait set.

  // TODO(chrisha): A quick code path for getting a context for a trait set that is changed
  // along a single dimension, as this is the most common operation (pushing a new trait,
  // or popping an existing trait).
};
```

Contexts would be attached to *base::TaskRunner, scheduler::TaskQueue* objects and *base::Location* objects. Setting a context on a TaskRunner would make it the default context for every task posted to it. Attaching a specific context to a Location object would override the default TaskRunner context, if one was provided. Location objects would also automatically carry the *parent context* (the context that was active at the time of the call to PostTask.

The implementation of the TaskRunner would be responsible for determining the active and parent contexts for a particular task (querying the TaskRunner and the Location object), and setting the parent and active contexts before commencing execution of a given task.

Propagating Traits

Across Task Boundaries

It is possible to automatically propagate traits across task boundaries by always attaching the current context to the Location object in every call to PostTask. FrameBlamr experimented with this but discovered that traits would quickly propagate to color the entire system. As such, automatic propagation is likely not the right solution. A compromise is to propagate the active context at the time of PostTask via a *parent* context side-channel. A task can then choose to inherit its parent's context very easily.

Across Process Boundaries

It is desirable to propagate traits seamlessly across IPC boundaries. Invoking a Mojo method with an active context should cause that context to be available as the parent context in the process where the work is actually being performed.

Synchronization and Consistency

It will be too expensive for this system to synchronous. As such, the focus should be on quick local operations and eventual consistency with the central service.

There are three sets of data that need to eventually be reconciled across process boundaries:

1. Trait objects belonging to dynamic trait classes.
2. Hashed IDs of trait sets.
3. Aggregates.

Synchronizing Trait Objects

It is possible and even likely that individual processes will have distinct sets of active objects in a given trait class. As such, it's burdensome to force a single process to maintain the union of that entire set. Even more so given the fact that the objects themselves are only needed during inspection and aggregation, which are driven by explicit user actions. As such, cross-process aggregation need not be real-time and any reconciliation can be deferred. Trait objects can be dealt with via a relatively simple mechanism.

- Object IDs are never reused for a given trait class in a given process.

- Each process maintains its own distinct set of object IDs and associated serialized objects.
- Each process maintains a lock-free description of object IDs, their associated class IDs, and their serialized forms. This should be in a format that is easy to read from a remote process, and be safe to read while simultaneously being modified.
- When reconciliation needs to happen the central service iterates over each child's active oject ID information, and constructs a map for translating remote object IDs to local object IDs. The child process map can contain locations for the central process to write the equivalent central service object IDs saving this calculation from being repeated.

The central process can make its object IDs available in a read-only manner to all child processes. This allows children to independently translate central object IDs to local object IDs on demand. It is anticipated that this will only need to happen during translation of traits embedded in IPC messages. The child process can then update the central-object-ID in its shared memory map of local object IDs, saving the parent process from repeated calculation during aggregation.

Synchronizing Trait Sets

A similar mechanism allows trait sets to be synchronized across processes.

- Each process maintains a list of active contexts (trait sets and their associated aggregates) in a lock-free shared memory buffer. This should be in a format that is easy to read from a remote process, and be safe to read while simultaneously being modified.
- Once a child to parent object ID mapping has been generated then the trait sets can be iterated and converted to local trait sets / trait set IDs. The child process map can contain shared locations for the central process to write the equivalent central service object IDs savings this calculation from being repeated.

Similar to object IDs, the central process can export its trait sets to children via shared memory. This allows children to translate central-IDs to local-IDs. They should also update their local translation cache, saving the parent process from having to perform the calculation again during aggregation.

Rolling Up Aggregates

Once object IDs and trait set IDs are synchronized each aggregate can be properly rolled up with identical contexts in other processes. While there is the possibility that trait sets have been modified or new one created since the central process performed the iteration, the data structures allow the iteration to proceed safely. In the worst case, the aggregation will be slightly behind real-time. However, eventual consistency is guaranteed.

Task Runner Integration

In order for activity traits to fulfill its goal it needs to be integrated into all of the various task runner systems in Chrome.

- Using base::Location provides integration into base::MessageLoop, base::ThreadTaskRunnerHandle, and base::TaskRunner. base::TaskRunner can be extended directly to support associated with it a "default" ActivityContext.
- Each scheduler::TaskQueue can be extended to have associated with it a default ActivityContext.
- Blink's tasks will generally be annotated by virtue of being posted to an already labeled scheduler::TaskQueue. However, Blink's WebTaskRunner could be directly extended to support a default ActivityContext. Doing this might impact exactly where the ActivityTraits code can live, and certainly limits the dependencies it can take.
- Completely transparent support can theoretically be integrated directly into Mojo, but is not likely and is beyond the scope of the initial implementation. It is more likely that individual messages would be

Unifying Trait Mechanisms

There are various trait mechanisms already in place. This aims to replace and unify them with a single coherent system.

- FrameBlamer
- V8 Tracing
- AllocationContext (memory-infra)