mozci

Release 1.2.8

Jul 28, 2023

Contents:

Mozci is an object oriented library aimed at making it easier to analyze pushes and tasks in Mozilla's CI system. Basic usage involves instantiating a `Push` object then accessing the attributes and calling the functions to retrieve the desired data of this push. For example:

```
from mozci.push import Push

push = Push("79041cab0cc2", branch="autoland")
print("\n".join([t.label for t in push.tasks if t.failed])
```

The above snippet prints the failed tasks for a given push. Mozci uses data from a variety of sources, including Treeherder, hg.mozilla.org and decision task artifacts.

See the *API docs* for more details.

CHAPTER 1

Usage Tips

Below are some ways to get the best experience with `mozci`.

1.1 Caching Results

Gathering the requisite data can sometimes be very expensive. Analyzing many pushes at once can take hours or even days. Luckily, `mozci` uses a caching mechanism so once a result is computed once, it won't be re-computed (even between runs). See the linked docs for more details, but a basic file system cache can be set up by modifying `~/.config/mozci/config.toml` and adding the following:

```
[mozci.cache]
retention = 10080  # minutes

[mozci.cache.stores]
file = { driver = "file", path = "/path/to/cache" }
```

1.1.1 Pre-seeding the Cache via Bugbug

There's a service called bugbug that runs `mozci` against all of the pushes on autoland. This service uploads its cache on S3 for others to use. You can benefit by using this uploaded cache to "pre-seed" your own local cache, if you have the necessary scopes. To do so, add the following to your `~/.config/mozci/config.toml`:

```
[mozci.cache]
serializer = "compressedpickle"

[mozci.cache.stores]
s3 = { driver = "s3", bucket = "communitytc-bugbug", prefix = "data/mozci_cache/" }
```

CHAPTER 2

Configuration

Mozci looks for a configuration file in your user config dir (e.g `~/.config/mozci/config.toml`):

The config is a TOML file, which looks something like:

```
[mozci]
verbose = 1
```

2.1 List of Options

The following keys are valid config options.

2.1.1 cache

This value allows you to set up a cache to store the results for future use. This avoids the penalty of hitting expensive data sources.

The `mozci` module uses cachy to handle caching. Therefore the following stores are supported:

- database
- file system
- memcached
- redis

To enable caching, you'll need to configure at least one store using the `cache.stores` key. Follow cachy's configuration format identically. In addition to the options `cachy` supports, you can set the `mozci.cache.retention` key to the time in minutes before stored queries are invalidated.

For example:

```
[mozci.cache]
retention = 10080  # minutes

[mozci.cache.stores]
file = { driver = "file", path = "/path/to/dir/to/keep/cache" }
```

In addition, `mozci` defines several custom cache stores:

- a `seeded-file` store. This is the same as the "file system" store, except you can specify a URL to initially seed your cache on creation:

```
[mozci.cache.stores]
file = {
    driver = "seeded-file",
    path = "/path/to/dir/to/keep/cache",
    url = "https://example.com/mozci_cache.tar.gz"
}
```

Supported archive formats include `.zip`, `.tar`, `.tar.gz`, `.tar.bz2` and `.tar.zst`.

The config also accepts a `reseed_interval` (in minutes) which will re-seed the cache after the interval expires. This assumes the URL is automatically updated by some other process.

As well as an `archive_relpath` config, which specifies the path to the cache data "within" the archive. Otherwise the cache data is assumed to be right at the root of the archive.

- a `renewing-file` store. This is the same as the "file system" store, except it renews items in the cache when they are retrieved.
- an `s3` store, which allows caching items in a S3 bucket. With this store, items are renewed on access like `renewing-file`. It's suggested to use a S3 Object Expiration policy to clean up items which are not accessed for a long time. Example configuration:

```
[mozci.cache.stores]
s3 = {
    driver = "s3",
    bucket = "myBucket",
    prefix = "data/mozci_cache/"
}
```

2.1.2 data_sources

Mozci can retrieve data from many different sources, e.g treeherder, taskcluster, hg.mozilla.org, etc. Often these sources can provide the same data, but may have different runtime characteristics. For example, some may not have realtime data, might require authentication or might take a really long time.

You can choose which sources you want to use with this key. For example:

```
[mozci]
data_sources = ["treeherder_client", "taskcluster"]
```

The above will first try to fulfill any data requirements using the `treeherder_client` source. But if that source is unable to fulfill the contract, the `taskcluster` source will be used as a backup.

Available sources are defined in the `DataHandler` class.

2.1.3 verbose

Enable verbose logging (default: `0`). Setting this to `1` enables debug logging, while setting it to `2` enables trace logging.

2.1.4 autoclassification

Mozci controls which push classification results can be automatically processed by third-party tools (like Treeherder), using a feature flag and a set of filters for test names.

The feature can be fully disabled by setting *enabled* to *false*.

```
[mozci.autoclassification]
enabled = true
test-suite-names = [
  "test-linux64-*/opt-mochitest-*",
  "*wpt*",
]
```

Each value in the list of *test-suite-names* support [fnmatch](https://docs.python.org/3/library/fnmatch.html#fnmatch.fnmatch) syntax to allow glob-like syntax (using * for wildcard and *?* for single characters).

The configuration above will enable autoclassification for tests matching *test-linux64-*/opt-mochitest-** or **wpt**

Finally, the JSON classification output is extended to have an *autoclassify* boolean flag on each failure details payload, to check if this specific result should be processed.

CHAPTER 3

Regressions

One of the primary uses of `mozci` is to help detect which tasks and/or tests (if any) a push has regressed. Since we do not run all tasks on every push and because of other factors like intermittents, this problem is more difficult than it first appears. In fact `mozci` can make very few guarantees and so has to rely on probabilistic guesses.

This page will help explain how regressions are calculated by introducing concepts one at a time.

3.1 Definitions

There are currently two different vectors of regression that `mozci` can check for: *label* and *group*.

- **label** - is a task label (e.g `test-linux1804-64/debug-mochitest-e10s-1`)
- **group** - is a grouping of tests, typically a manifest (e.g `dom/indexedDB/test/mochitest.ini`).
- **runnable** is the unique label identifying a set of tasks, or the unique group identifying a set of tests.
- **classification** - an annotation that Sheriffs apply to tasks manually. It is also known as "starring" because it puts a little asterisk next to the task in Treeherder.

3.2 Runnable Summary

Thanks to retriggers, each runnable can run multiple times on the same push. The collection of labels or groups of the same type that ran on a push is called a *runnable summary*. For instance, if all the runnables on a push passed, then the status of the runnable summary is also PASS. Likewise if they all failed. If at least one instance of a runnable passes, and at least one instance of a runnable failed, then the runnable summary is said to be intermittent.

The `GroupSummary` class implements this logic for groups and the `LabelSummary` implements the logic for labels. Both classes inherit from the `RunnableSummary` abstract base class.

All instances of `RunnableSummary` have an overall status and an overall classification.

3.3 Candidate Regression

A candidate regression is a runnable which meets the following criteria:

- At least one instance of this runnable failed on target push (i.e, the status of the runnable summary is either FAIL or INTERMITTENT)
- The overall classification of the runnable summary is either `unclassified`, or `fixed by commit`. This means runnables classified as a known intermittent are not candidate regressions.
- For runnables classified `fixed by commit`, the referenced backout backs out the target push and not some other one.

OR

- The runnable ran on a child push (up to `MAX_DEPTH` pushes away), and is classified `fixed by commit`.
- The classification references a backout that backs out the target push.

Candidate regressions are the set of all runnables that could possibly be a regression of this push. This *does not* mean that they are regressions. Just that they could be.

The set of candidate regressions can be obtained by calling `Push.get_candidate_regressions()`.

3.4 Regression

A *regression* is a candidate regression that additionally satisfies the following criteria:

- The candidate regression is not marked as a regression of any parent pushes up to `MAX_DEPTH` pushes away.
- The condition `total_distance <= MAX_DEPTH` is satisfied. This condition is explained in more detail below.

Note: Distance Calculation

The `total_distance` is the number of parent pushes we need to go back to see the runnable plus the number of child pushes we need to go forward to see the runnable. A `total_distance` of 0 means the runnable ran on the actual target push.

The `total_distance` can be modified in certain scenarios:

1. The push was not backed out => total distance is doubled.
2. The runnable was intermittent => total distance is doubled.
3. The runnable was marked as `fixed by commit` referencing a backout that backs out the target push => total distance is 0 even if it didn't run on the target push.

These modifications help us deal with (un)certainty in special easy to detect circumstances. The first two make a candidate regression less likely to be treated as a regression, while the third guarantees it.

Regressions can be obtained by calling `Push.get_regressions()`.

3.5 Likely Regressions

A *likely regression* is a regression whose associated `total_distance` is 0. In other words, we are as sure as we can be that these are regressions.

Likely regressions can be obtained by calling `Push.get_likely_regressions()`.

3.6 Possible Regressions

A *possible regression* is a regression whose associated `total_distance` is above 0. In other words, it could be a regression, or it could be regressed from one of its parent pushes. We aren't sure. The higher the `total_distance` the less sure we are.

Possible regressions can be obtained by calling `Push.get_possible_regressions()`.

Note: Candidate regressions that aren't also possible regressions could still technically be real regressions. Mozci just thinks the likelihood is so low they aren't worth counting.

CHAPTER 4

mozci

4.1 mozci package

4.1.1 Subpackages

mozci.data package

Subpackages

mozci.data.sources package

Subpackages

mozci.data.sources.hgmo package

Module contents

mozci.data.sources.taskcluster package

Module contents

mozci.data.sources.treeherder package

Module contents

Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search