Bazaar Architecture Overview

Release 2.7.1dev

Bazaar Developers

March 22, 2019

Contents

This document describes the key classes and concepts within Bazaar. It is intended to be useful to people working on the Bazaar codebase, or to people writing plugins. People writing plugins may also like to read the guide to Integrating with Bazaar for some specific recipes.

There's some overlap between this and the Core Concepts section of the user guide, but this document is targetted to people interested in the internals. In particular the user guide doesn't go any deeper than "revision", because regular users don't care about lower-level details like inventories, but this guide does.

If you have any questions, or if something seems to be incorrect, unclear or missing, please talk to us in `irc://irc.freenode.net/#bzr`, write to the Bazaar mailing list, or simply file a bug report.

1 IDs and keys

1.1 IDs

All IDs are globally unique identifiers. Inside bzrlib they are almost always represented as UTF-8 encoded bytestrings (i.e. `str` objects).

The main two IDs are:

Revision IDs The unique identifier of a single revision, such as `pqm@pqm.ubuntu.com-20110201161347-ao76mv267gc1b5v2`

File IDs The unique identifier of a single file.

By convention, in the bzrlib API, parameters of methods that are expected to be IDs (as opposed to keys, revision numbers, or some other handle) will end in `id`, e.g. `revid` or `file_id`.

Ids may be stored directly or they can be inferred from other data. Native Bazaar formats store ids directly; foreign VCS support usually generates them somehow. For example, the Git commit with SHA `fb235a3be6372e40ff7f7ebbcd7905a08cb04444` is referred to with the revision ID `git-v1:fb235a3be6372e40ff7f7ebbcd7905a08cb04444`. IDs are expected to be persistent

File ids

File ids are unique identifiers for files. There are three slightly different categories of file ids.

Tree file ids

Tree file ids are used in the `Tree` API and can either be UTF-8 encoded bytestrings or tuples of UTF-8 encoded bytestrings. Plain bytestrings are considered to be the equivalent of a 1-tuple containing that bytestring.

Tree file ids should be considered valid only for a specific tree context. Note that this is a stricter interpretation than what the current bzr format implementation provides - its file ids are persistent across runs and across revisions.

For some formats (most notably bzr's own formats) it's possible for the implementation to specify the file id to use. In other case the tree mandates a specific file id.

Inventory file ids

Inventories are specific to the bzr native format and are the main layer below the `Tree` implementation of bzr. File ids in inventories can only be UTF-8 encoded bytestrings. A single Tree object can be associated with multiple inventories if there are nested trees.

Tree file ids for bzr formats are a tuple of inventory file ids for the file in question. Each non-last item in the tuple refers to the tree reference of an inner tree. The last item in the tuple refers to the actual file. This means that lookups of file ids doesn't scale with the number of nested trees.

Inventory file ids are only relevant for native Bazaar formats; foreign formats don't use inventories.

Transform ids

Transform ids are used during tree transform operations (used by e.g. merge). The same transform id is expected to be used for two instances of the same file. At the moment transform ids are directly derived from file ids, but in the future they could be based on other data too (e.g. automatic rename detection or format-specific rules).

1.2 Keys

A composite of one or more ID elements. E.g. a (file-id, revision-id) pair is the key to the "texts" store, but a single element key of (revision-id) is the key to the "revisions" store.

2 Core classes

2.1 Transport

The `Transport` layer handles access to local or remote directories. Each Transport object acts as a logical connection to a particular directory, and it allows various operations on files within it. You can *clone* a transport to get a new Transport connected to a subdirectory or parent directory.

Transports are not used for access to the working tree. At present working trees are always local and they are accessed through the regular Python file I/O mechanisms.

Filenames vs URLs

Transports work in terms of URLs. Take note that URLs are by definition only ASCII - the decision of how to encode a Unicode string into a URL must be taken at a higher level, typically in the Store. (Note that Stores also escape filenames which cannot be safely stored on all filesystems, but this is a different level.)

The main reason for this is that it's not possible to safely roundtrip a URL into Unicode and then back into the same URL. The URL standard gives a way to represent non-ASCII bytes in ASCII (as %-escapes), but doesn't say how those bytes represent non-ASCII characters. (They're not guaranteed to be UTF-8 – that is common but doesn't happen everywhere.)

For example, if the user enters the URL `http://example/%e0`, there's no way to tell whether that character represents "latin small letter a with grave" in iso-8859-1, or "latin small letter r with acute" in iso-8859-2, or malformed UTF-8. So we can't convert the URL to Unicode reliably.

Equally problematic is if we're given a URL-like string containing (unescaped) non-ASCII characters (such as the accented a). We can't be sure how to convert that to a valid (i.e. ASCII-only) URL, because we don't know what encoding the server expects for those characters. (Although it is not totally reliable, we might still accept these and assume that they should be put into UTF-8.)

A similar edge case is that the URL `http://foo/sweet%2Fsour` contains one directory component whose name is "sweet/sour". The escaped slash is not a directory separator, but if we try to convert the URL to a regular Unicode path, this information will be lost.

This implies that Transports must natively deal with URLs. For simplicity they *only* deal with URLs; conversion of other strings to URLs is done elsewhere. Information that Transports return, such as from `list_dir`, is also in the form of URL components.

More information

See also:

- Developer guide to bzrlib transports
- API docs for `bzrlib.transport.Transport`

2.2 Control directory

Each control directory (such as ".bzr/") can contain zero or one repositories, zero or one working trees and zero or more branches.

The `BzrDir` class is the `ControlDir` implementation that is responsible for the ".bzr/" directory and its implementation. Plugins that provide support for other version control systems can provide other subclasses of `ControlDir`.

2.3 Tree

A representation of a directory of files (and other directories and symlinks etc). The most important kinds of Tree are:

WorkingTree the files on disk editable by the user

RevisionTree a tree as recorded at some point in the past

Trees can map file paths to file-ids and vice versa (although trees such as WorkingTree may have unversioned files not described in that mapping). Trees have an inventory and parents (an ordered list of zero or more revision IDs).

The implementation of `Tree` for Bazaar's own formats is based around `Inventory` objects which describe the shape of the tree. Each tree has at least one inventory associated with it, which is available as the `root_inventory` attribute on tree. The tree can have more inventories associated with it if there are references to other trees in it. These references are indicated with `tree-reference` inventory entry at the point where the other tree is nested. The tree reference entry contains sufficient information for looking up the inventory associated with the nested tree. There can be multiple layers of nesting.

Not each `Tree` implementation will necessarily have an associated `root_inventory`, as not all implementations of `Tree` are based around inventories (most notably, implementations of foreign VCS file formats).

2.4 WorkingTree

A workingtree is a special type of Tree that's associated with a working directory on disk, where the user can directly modify the files.

Responsibilities:

- Maintaining a WorkingTree on disk at a file path.
- Maintaining the basis inventory (the inventory of the last commit done)
- Maintaining the working inventory.
- Maintaining the pending merges list.
- Maintaining the stat cache.
- Maintaining the last revision the working tree was updated to.
- Knows where its Branch is located.

Dependencies:

- a Branch
- local access to the working tree

2.5 Branch

A Branch is a key user concept - its a single line of history that one or more people have been committing to.

A Branch is responsible for:

- Holding user preferences that are set in a Branch.
- Holding the 'tip': the last revision to be committed to this Branch. (And the revno of that revision.)
- Knowing how to open the Repository that holds its history.
- Allowing write locks to be taken out to prevent concurrent alterations to the branch.

Depends on:

- URL access to its base directory.
- A Transport to access its files.
- A Repository to hold its history.

2.6 Repository

Repositories store committed history: file texts, revisions, inventories, and graph relationships between them. A repository holds a bag of revision data that can be pointed to by various branches:

- Maintains storage of various history data at a URL:
  - Revisions (Must have a matching inventory)
  - Digital Signatures
  - Inventories for each Revision. (Must have all the file texts available).
  - File texts
- Synchronizes concurrent access to the repository by different processes. (Most repository implementations use a physical mutex only for a short period, and effectively support multiple readers and writers.)

Stacked Repositories

A repository can be configured to refer to a list of "fallback" repositories. If a particular revision is not present in the original repository, it refers the query to the fallbacks.

Compression deltas don't span physical repository boundaries. So the first commit to a new, empty repository with fallback repositories will store a full text of the inventory, and of every new file text.

At runtime, repository stacking is actually configured by the branch, not the repository. So doing `a_bzrdir.open_repository()` gets you just the single physical repository, while `a_bzrdir.open_branch().repository` gets one configured with a stacking. Therefore, to permanently change the fallback repository stored on disk, you must use `Branch.set_stacked_on_url`.

Changing away from an existing stacked-on URL will copy across any necessary history so that the repository remains usable.

A repository opened from an HPSS server is never stacked on the server side, because this could cause complexity or security problems with the server acting as a proxy for the client. Instead, the branch on the server exposes the stacked-on URL and the client can open that.

3 Storage model

This section describes the model for how bzr stores its data. The representation of that data on disk varies considerable depending on the format of the repository (and to a lesser extent the format of the branch and working tree), but ultimately the set of objects being represented is the same.

3.1 Branch

A branch directly contains:

- the ID of the current revision that branch (a.k.a. the "tip")
- some settings for that branch (the values in "branch.conf")
- the set of tags for that branch (not supported in all formats)

A branch implicitly references:

- A repository. The repository might be colocated in the same directory as the branch, or it might be somewhere else entirely.

3.2 Repository

A repository contains:

- a revision store
- an inventory store
- a text store
- a signature store

A store is a key-value mapping. This says nothing about the layout on disk, just that conceptually there are distinct stores, each with a separate namespace for the keys. Internally the repository may serialize stores in the same file, and/or e.g. apply compression algorithms that combine records from separate stores in one block, etc.

You can consider the repository as a single key space, with keys that look like *(store-name, ...)*. For example, *('revisions', revision-id)* or *('texts', revision-id, file-id)*.

Revision store

Stores revision objects. The keys are GUIDs. The value is a revision object (the exact representation on disk depends on the repository format).

As described in Core Concepts a revision describes a snapshot of the tree of files and some metadata about them.

- metadata:
  - parent revisions (an ordered sequence of zero or more revision IDs)
  - commit message
  - author(s)

  - timestamp
  - (and all other revision properties)
- an inventory ID (that inventory describes the tree contents). Is often the same as the revision ID, but doesn't have to be (e.g. if no files were changed between two revisions then both revisions will refer to the same inventory).

Inventory store

Stores inventory objects. The keys are GUIDs. (Footnote: there will usually be a revision with the same key in the revision store, but there are rare cases where this is not true.)

An inventory object contains:

- a set of inventory entries

An inventory entry has the following attributes

- a file-id (a GUID, or the special value TREE_ROOT for the root entry of inventories created by older versions of bzr)
- a revision-id, a GUID (generally corresponding to the ID of a revision). The combination of (file-id, revision-id) is a key into the texts store.
- a kind: one of file, directory, symlink, tree-reference (tree-reference is only supported in unsupported developer formats)
- parent-id: the file-id of the directory that contains this entry (this value is unset for the root of the tree).
- name: the name of the file/directory/etc in that parent directory
- executable: a flag indicating if the executable bit is set for that file.

An inventory entry will have other attributes, depending on the kind:

- file:
  - SHA1
  - size
- directory
  - children
- symlink
  - symlink_target
- tree-reference
  - reference_revision

For some more details see Inventories.

Texts store

Stores the contents of individual versions of files. The keys are pairs of (file-id, revision-id), and the values are the full content (or "text") of a version of a file.

For consistency/simplicity text records exist for all inventory entries, but in general only entries with of kind "file" have interesting records.

Signature store

Stores cryptographic signatures of revision contents. The keys match those of the revision store.