HashDist Documentation

Release 0.3

HashDist team

May 08, 2016

Contents

These documents are a work in progress. Some may be out of date. Please file an issue if the instructions appear to be incorrect.

CHAPTER 1

Working with HashDist

1.1 User's guide to HashDist

1.1.1 Installing and making the *hit* tool available

HashDist requires Python 2.7 and git.

To start using HashDist, clone the repo that contains the core tool, and put the `bin`-directory in your `PATH`:

```
$ git clone https://github.com/hashdist/hashdist.git
$ cd hashdist
$ export PATH=$PWD/bin:$PATH
```

The `hit` tool should now be available. You should now run the following command to create the directory `~/.hashdist`:

```
$ hit init-home
```

By default all built software and downloaded sources will be stored beneath `~/.hashdist`. To change this, edit `~/.hashdist/config.yaml`.

1.1.2 Setting up your software profile

Using HashDist is based on the following steps:

1. First, describe the software profile you want to build in a configuration file ("I want Python, NumPy, SciPy").
2. Use a dedicated git repository to manage that configuration file
3. For every git commit, HashDist will be able to build the specified profile, and *cache* the results, so that you can jump around in the history of your software profile.

Start with cloning a basic user profile template:

```
git clone https://github.com/hashdist/profile-template.git /path/to/myprofile
```

The contents of the repo is a single file `default.yaml` which a) selects a *base profile* to extend, and b) lists which packages to include. It is also possible to override build parameters from this file, or link to extra package descriptions within the repository (docs not written yet). The idea is to modify this repository to make changes to the software profile that only applies to you. You are encouraged to submit pull requests against the base profile for changes that may be useful to more users.

To build the stack, simply do:

```
cd /path/to/myprofile
hit build
```

This will take a while, including downloading the source code needed. In the end, a symlink `default` is created which contains the exact software described by `default.yaml`.

Now, try to remove the `jinja2` package from `default.yaml` and do `hit build` again. This time, the build should only take a second, which is the time used to assemble a new profile.

Then, add the `jinja2` package again and run `hit build`. This exact software profile was already built, and so the operation is very fast.

When coupled with managing the profile specification with git, this becomes very powerful, as you can use git to navigate the history of or branches of your software profile repository, and then instantly switch to pre-built versions. [TODO: `hit commit`, `hit checkout` commands.]

If you want to have, e.g., release and debug profiles, you can create `release.yaml` and `debug.yaml`, and use `hit build release.yaml` or `hit build debug.yaml` to select another profile than `default.yaml` to build.

1.1.3 Garbage collection

HashDist does not have the concepts of "upgrade" or "uninstall", but simply keeps everything it has downloaded or built around forever. To free up disk space, you may invoke the garbage collector to remove unused builds.

Currently the garbage collection strategy is very simple: When you invoke garbage collection manually, HashDist removes anything that isn't currently in use. To figure out what that means, you may invoke `hit gc --list`; continueing on the example from above, we would find:

```
$ hit gc --list
List of GC roots:
/path/to/myprofile/default
```

This indicates that if you run a plain `hit gc`, software accessible through `/path/to/myprofile/default` will be kept, but all other builds will be removed from the HashDist store. To try it, you may comment out the `zlib` line from `default.yaml`, then run `hit build`, and then `hit gc` – the zlib software is removed at the last step.

If you want to manipulate profile symlinks, you should use the `hit cp`, `hit mv`, and `hit rm` commands, so that HashDist can correctly track the profile links. This is useful to keep multiple profiles around. E.g., if you first execute:

```
hit cp default old_profile
```

and then modify `default.yaml`, and then run `hit build`, then after the build `default` and `old_profile` will point to different revisions of the software stacks, both usable at the same time. Garbage collection will keep software for either around.

The database of GC roots is kept (by default) in `~/.hashdist/gcroots`. You are free to put your own symlinks there (you may give them an arbitrary name, as long as they do not contain an underscore in front), or manually remove symlinks.

> **Warning:** As a corollary to the description above, if you do a plain `mv` of a symlink to a profile, and then execute `hit gc`, then the software profile may be deleted by HashDist.

1.1.4 Debug features

A couple of commands allow you to see what happens when building.

- Show the script used to build Jinja2:

```
hit show script jinja2
```

- Show the "build spec" (low-level magic):

```
hit show buildspec jinja2
```

- Get a copy of the build directory that would be used:

```
hit bdir jinja2 bld
```

This extracts Jinja2 sources to `bld`, puts a Bash build-script in `bld/_hashdist/build.sh`. However, if you go ahead and try to run it the environment will not be the same as when HashDist builds, so this is a bit limited so far. [TODO: `hit debug` which also sets the right environment and sets the `$ARTIFACT` directory.]

1.1.5 Developing the base profile

If you want to develop the `hashstack` repository yourself, using a dedicated local-machine profile repo becomes tedious. Instead, copy the `default.example.yaml` to `default.yaml`. Then simply run `hit build` directly in the base profile (in which case the personal profile is not needed at all).

`default.yaml` is present in `.gitignore` and changes should not be checked in; you freely change it to experiment with whatever package you are adding. Note the orthogonality between the two repositories: The base profile repo has commits like "Added build commands for NumPy 1.7.2 to share to the world". The personal profile repo has commits like "Installed the NumPy package on my computer".

1.1.6 Further details

Specifying a HashDist software profile

1.2 Specifying a HashDist software profile

There are specification file types in HashDist. The *profile spec* describes *what* to build; what packages should be included in the profile and the options for each package. A *package spec* contains the *how* part: A (possibly parametrized) description for building a single package.

The basic language of the specification files is YAML, see http://yaml.org. Style guide: For YAML files within the HashDist project, we use 2 space indents, and no indent before vertically-formatted lists (as seen below).

1.2.1 Profile specification

The profile spec is what the user points the *hit* tool to to build a profile. By following references in it, HashDist should be able to find all the information needed (including the package specification files). An example end-user profile might look like this:

```
extends:
- name: hashstack
  urls: ['https://github.com/hashdist/hashstack.git']
  key: 'git:5042aeaaee9841575e56ad9f673ef1585c2f5a46'
  file: debian.yaml

- file: common_settings.yaml
```

```
parameters:
  debug: false

packages:
  zlib:
  szlib:
  nose:
  python:
    host: true
  mpi:
    use: openmpi
  numpy:
    skip: true

package_dirs:
- pkgs
- base

hook_import_dirs:
- base
```

extends:

Profiles that this profile should extend from. Essentially this profile is merged on a parameter-by-parameter and package-by-package basis. If anything conflicts there is an error. E.g., if two base profiles sets the same parameter, the parameter must be specified in the descendant profile, otherwise it is an error.

There are two ways of importing profiles:

- **Local**: Only provide the **file** key, which can be an absolute path, or relative to the directory of the profile spec file.
- **Remote**: If **urls** (currently this must be a list of length one) and **key** are given, the specified sources (usually a git commit) will be downloaded, and the given **file** is relative to the root of the repo. In this case, providing a **name** for the repository is mandatory; the name is used to refer to the repository in error messages etc., and must be unique for the repository across all imported profile files.

parameters:

Global parameters set for all packages. Any parameters specified in the **packages** section will override these on a per-package basis.

Parameters are typed as is usual for YAML documents; variables will take the according Python types in expressions/hooks. E.g., `false` shows up as *False* in expressions, while `'false'` is a string (evaluating to *True* in a boolean context).

packages:

The packages to build. Each package is given as a key in a dict, with a sub-dict containing package-specific parameters. This is potentially empty, which means "build this package with default parameters". If a package is not present in this section (and is not a dependency of other packages) it will not be built. The **use** parameter makes use of a different package name for the package given, e.g., above, package specs for `openmpi` will be searched and built to satisfy the `mpi` package. The **skip** parameter says that a package should *not* be built (which is useful in the case that the package was included in an ancestor profile).

package_dirs:

Directories to search for package specification files (and hooks, see section on Python hook files below). These acts in an "overlay" manner. In the example above, if one e.g., if searching for

`python_package.yaml` then first the `pkgs` sub-directory relative to the profile file will be consulted, then `base`, and finally any directories listed in **package_dirs** in the base profiles extended in **extends**.

This way, one profile can override/replace the package specifications of another profile by listing a directory here.

The common case is that base profiles set **package_dirs**, but that overriding user profiles do not have it set.

hook_import_dirs:

Entries for `sys.path` in Python hook files. Relative to the location of the profile file.

1.2.2 Package specifications

Below we assume that the directory `pkgs` is a directory listed in **package_dirs** in the profile spec. We can then use:

- Single-file spec: `pkgs/mypkg.yaml`
- Multi-file spec: `pkgs/mypkg/mypkg.yaml`, `pkgs/mypkg/somepatch.diff`, `pkgs/mypkg/mypkg-linux.yaml`

In the latter case, all files matching `mypkg/mypkg.yaml` and `mypkg/mypkg-*.yaml` are loaded, and the **when** clause evaluated for each file. An error is given if more than one file matches the given parameters. One of the files may lack the **when** clause (conventionally, the one without a dash and a suffix), which corresponds to a default fallback file.

Also, HashDist searches in the package directories for `mypkg.py`, which specifies a Python module with hook functions that can further influence the build. Documentation for the Python hook system is TBD, and the API tentative. Examples in `base/autotools.py` in the Hashstack repo.

Examples of package specs are in https://github.com/hashdist/hashstack, and we will not repeat them here, but simply list documentation on each clause.

In strings; `{{param_name}}` will usually expand to the parameter in question while assembling the specification needed, and are expanded before artifact hashes are computed. Expansions of the form `${FOO}` are expanded at build-time (by the HashDist build system or the shell, depending on context), and the variable name is what is hashed.

when:

Conditions for using this package spec, see rules above. It is a Python expression, evaluated in a context where all parameters are available as variables

extends:

A list of package names. The package specs for these *base packages* will be loaded and their contents included, as documented below.

sources:

Sources to download. This should be a list of `key` and `url` pairs. To generate the `key` for a new file, use the `hit fetch` command.

dependencies:

Lists of names for packages needed during build (**build** sub-clause) or in the same profile (**run** sub-clause). Dependencies from base packages are automatically included in these lists, e.g., if `python_package` is listed in **extends**, then `python_package.yaml` may take care of requiring a build dependency on Python.

build_stages:

Stages for the build. See Stage system section below for general comments. The build stages are ordered and then executed to produce a Bash script to run to do the build; the **handler** attribute (which defaults to the value of the **name** attribute) determines the format of the rest of the stage.

when_build_dependency:

Environment variable changes to be done when this package is a build dependency for *another* package. As a special case variable `${ARTIFACT}`

profile_links:

A small DSL for setting up links when building the profile. What links should be created when assembling a profile. (In general this is dark magic and subject to change until documented further, but usually only required in base packages.)

1.2.3 Conditionals

The top-level **when** in each package spec has already been mentioned. In addition, there are two forms of local conditionals withi a file. The first one can be used within a list-of-dicts, e.g., in **build_stages** and similar sections:

```
- when: platform == 'linux'
  name: configure
  extra: [--with-foo]

- when: platform == 'windows'
  name: configure
  extra: [--with-bar]
```

The second form takes the form of a more traditional if-test:

```
- name: configure
  when platform == 'linux':
      extra: [--with-foo]
  when platform == 'windows':
      extra: [--with-bar]
  when platform not in ('linux', 'windows'):
      extra: [--with-baz]
```

The syntax for conditional list-items is a bit awkward, but available if necesarry:

```
dependencies:
  build:
    - numpy
    - when platform == 'linux':  # ! note the dash in front
      - openblas
    - python
```

This will turn into either `[numpy, python]` or `[numpy, openblas, python]`. The "extra" – is needed to maintain positioning within the YAML file.

1.2.4 Stage system

The **build_stages**, **when_build_dependency** and **profile_links** clauses all follow the same format: A list of "stages" that are partially ordered (using **name**, **before**, and **after** attributes). Thus one can inherit a set of stages from the base packages, and only override the stages one needs.

There's a special **mode** attribute which determines how the override happens. E.g.,:

```
- name: configure
  mode: override  # !!
  foo: bar
```

will pass an extra `foo: bar` attribute to the configure handler, in addition to the attributes that were already there in the base package. This is the default behaviour. On the other hand,:

```
- name: configure
  mode: replace  # !!
  handler: bash
  bash: |
      ./configure --prefix=${ARTIFACT}
```

entirely replaces the configure stage of the base package.

The `update` mode will update dictionaries and lists within a stage, so it can be helpful for building up a set of actions for a given stage,:

```
- name: configure
   append: {overriden_value: "1", a_key: "a"}
   extra: ['--shared']
- name: configure
  mode: update
  append: {overriden_value: "2", b_key: "b"}
  extra: ['--without-ensurepip']
```

is equivalent to,:

```
- name: configure
  append: {overriden_value: "2", a_key: "a", b_key: "b"}
  extra: ['--shared', '--without-ensurepip']
```

Finally,:

```
- name: configure
  mode: remove  # !!
```

removes the stage.

1.3 Guidelines for packaging/building

HashDist can be used to build software in many ways. This document describes what the HashDist authors recommend and the helper tools available for post-processing.

1.3.1 Principles

- Build artifacts should be *fully relocatable*, simply because we can, and it is convenient. In particular it means we can avoid adding complexity (an extra post-processing phase) for binary artifacts.
- It should be possible to use the software with as few environment variable modifications as possible. I.e., one should be able to do `path/to/profile/bin/program` without any changes to environment variables at all, and a typical setup would only modify `$PATH`.

1.3.2 Relocatable artifacts

Relocatability is something that must be dealt with on a case-by-case basis.

Artifact relative references

For relative artifacts to work (at least without serious patching of all programs involved...), it is currently necessarry to insist that all build artifacts involved are in the sub-directory/partition, so that relative paths such as `../../../python/34ssdf32/lib/..` remain valid.

Note: *Considering the future:* If multiple artifact dirs are needed, a possibility for splitting up build artifact repositories would be to symlink between them at the individual artifact level, say:

```
/site/hit/zlib/afda/...
/site/hit/python/fdas/...
```

And then another artifact directory could contain:

```
/home/dagss/.hit/opt/zlib/afda -> /site/hit/zlib/afda
/home/dagss/.hit/opt/python/fdas -> /site/hit/python/fdas
/home/dagss/.hit/opt/pymypackage/3dfs/...
```

Because artifacts should be a DAG this should work well. This could be naturally implemented as whenever a cached build artifact is found on a locally available filesystem, symlink to it.

Of course, now `/site/hit` is *not* relocatable, but mere rewriting of those symlinks, always at the same level in the filesystem, is a lot more transparent than full post-processing of artifacts.

Unix: Scripts

The shebang lines `#!/usr/bin/env interpreter` or `#!/full/path/to/interpreter` are limited and preclude relocatability by themselves. We deal with this by using multi-line shebangs.

We want to search for interpreters for scripts as follows:

- In the `bin`-dir of the profile in use (typically, but not always, along-side the script). We define this as: For each symlink in the chain between `$0` and the real script file, search upwards towards the root for a file named `profile.json`. If found, find the interpreter (by its base name) in the `bin`-subdirectory.
- Otherwise, fall back to the *relative* path between the real location of the script file and the interpreter's build artifact. (E.g., for Python, this could work if `PYTHONPATH` is also set correctly, as may be the case during builds.)

Example: The command:

```
$ hit build-postprocess --shebang=multiline path-or-script
```

applies the multiline shebang. Test script:

```
#!/tmp/../usr/bin/python

"""simple example"""

print 'Hello world'
```

Note that shebangs with `/usr/bin/*` is not processed, so we had to force the tool to kick in using `/tmp/../usr/bin/` (because the intention is really just to patch references to other artifacts). Then calling the command above yields:

```
#!/bin/sh
# -*- mode: python -*-
"true" '''\';r="../../../../usr/bin";i="python";o=`pwd`;p="$0";while true; do test -L "$p";il=$?;cd
```

```
''' # end multi-line shebang, see hashdist.core.build_tools

__doc__ = """simple example"""

print 'Hello world'
# vi: filetype=python
```

This file is executable both using Python 2.x or a POSIX shell. See the code of `hashdist.core.build_tools` for the shell script in a non-compacted form with comments. Note the `r` and `i` variables; `../../../../usr/bin` was simply the relative path between the example script and `/usr/bin` when the command was run.

Unix: Dynamic libraries

Modern Unix platforms allows binaries to link to dynamic libraries in relative locations by using an RPATH containing the string `${ORIGIN}`. See `man ld.so` (on Linux at least).

Passing this is almost impossible because whenever one uses the configure system it tends to add the absolute RPATH anyway. Also, the contortions one must go through to properly escape the magic string (`$` unfortunately being part of it) is build-system specific, for autoconf with libtool it is `\$$ORIGIN`, where first Make sees `$$`, and then Bash sees `\$`.

Fortunately, for Linux there is patchelf and for Mac OS X [another tool].

CHAPTER 2

Developer Resources

2.1 Developers's guide

2.1.1 Terminology

Distribution: An end-user software distribution that makes use of HashDist under the hood, e.g., python-hpcmp

Artifact: The result of a build process, identified by a hash of the inputs to the build

Profile: A "prefix" directory structure ready for use through `$PATH`, containing subdirectories `bin`, `lib`, and so on with all/some of the software one wants to use.

Package: Used in the loose sense; a program/library, e.g., NumPy, Python etc.; what is **not** meant is a specific package format like `.spkg`, `.egg` and so on (which is left undefined in the bottom two HashDist layers)

2.1.2 Design principles

- Many small components with one-way dependencies, accessible through command line and as libraries (initially for Python, but in principle for more languages too).
- Protocols between components are designed so that one can imagine turning individual components into server applications dealing with high loads.
- However, implementations of those protocols are currently kept as simple as possible.
- HashDist is a language-neutral solution; Python is the implementation language chosen but the core tools can (in theory) be rewritten in C or Ruby without the users noticing any difference
- The components are accessible through a common `hit` command-line tool. This accesses both power-user low-level features and the higher-level "varnish", without implying any deep coupling between them (just like *git*).

2.1.3 Powerusers' guide, layer by layer

HashDist consists of two (eventually perhaps three) layers. The idea is to provide something useful for as many as possible. If a distribution only uses the the *core layer* (or even only some of the components within it) it can keep on mostly as before, but get a performance boost from the caching aspect. If the distribution wants to buy into the greater HashDist vision, it can use the *profile specification layer*. Finally, for end-users, a final user-interface layer is needed to make things friendly. Here HashDist will probably remain silent for some time, but some standards, best practices and utilities may emerge. For now, a user interface ideas section is included below.

2.2 Build specifications and artifacts

2.2.1 The build spec (`build.json`)

The heart of HashDist are *build specs*, JSON documents that describes everything that goes into a build: the source code to use, the build environment, the build commands.

A *build artifact* is the result of executing the instruction in the build spec. The *build store* is the collection of locally available build artifacts.

The way the build store is queried is by using build specs as keys to look up software. E.g., take the following `build.json`:

```
{
    "name": "zlib",
    "sources": [
        {"key": "tar.gz:7kojzbry564mxdxv4toviu7ekv2r4hct", "target": ".", "strip": 1}
    ],
    "build": {
        "import": [
            {"ref": "UNIX", "id": "virtual:unix"},
            {"ref": "GCC", "id": gcc/gxuzlqihu4ok5obtxm5xt6pvi6a3gp5b"},
        ],
        "commands": [
            {"cmd": ["./configure", "--prefix=$ARTIFACT"]},
            {"cmd": ["make", "install"]}
        ]
    }
}
```

The *artifact ID* derived from this build specification is:

```
$ hit hash build.json
zlib/d4jwf2sb2g6glprsdqfdpcracwpzujwq
```

Let us check if it is already in the build store:

```
$ hit resolve build.json
(not built)
```

```
$ hit resolve -h zlib/d4jwf2sb2g6glprsdqfdpcracwpzujwq
(not built)
```

In the future, we hope to make it possible to automatically download build artifacts that are not already built. For now, building from source is the only option, so let's do that:

```
$ hit build -v build.json
...<output>...
/home/dagss/.hdist/opt/zlib/d4jw
```

The last line is the location of the corresponding *build artifact*, which can from this point of be looked up by the either the build spec or the hash:

```
$ hit resolve build.json
/home/dagss/.hdist/opt/zlib/d4jw
```

```
$ hit resolve -h zlib/d4jwf2sb2g6glprsdqfdpcracwpzujwq
/home/dagss/.hdist/opt/zlib/d4jw
```

Note: Note that the intention is not that end-users write `build.json` files themselves; they are simply the *API for the build artifact store*. It is the responsibility of distributions such as *python-hpcmp* to properly generate build specifications for HashDist in a user-friendly manner.

2.2.2 Build artifacts

A build artifact contains the result of a build; usually as a prefix-style directory containing one library/program only, i.e., typical subdirectories are `bin`, `include`, `lib` and so on. The build artifact should ideally be in a relocatable form that can be moved around or packed and distributed to another computer (see Guidelines for packaging/building), although HashDist does not enforce this.

A HashDist artifact ID has the form `name/hash`, e.g., `zlib/4niostz3iktlg67najtxuwwgss5vl6k4`. For the artifact paths on disk, a shortened form (4-char hash) is used to make things more friendly to the human user. If there is a collision, the length is simply increased for the one that comes later (see also *Discussion*).

Some information is present in every build artifact:

`build.log.gz`: The log from performing the build.

`build.json`: The input build spec.

`artifact.json`: Metadata about the build artifact itself. Some of this is simply copied over from `build.json`; however, this is a separate file because parts of it could be generated as part of the build process. This is important during **Profile generation** (docs TBD).

2.2.3 Build spec spec

The build spec document has the following top-level fields:

name: Used as the prefix in the artifact ID. Should match `[a-zA-Z0-9-_+]+`.

version: (Currently not used, but will become important for virtual build artifacts). Should match `[a-zA-Z0-9-_+]*`.

build: A *job* to run to perform the build. See *`hashdist.core.run_job`* for the documentation of this sub-document.

sources: Sources listed are unpacked to build directory; documentation for now in 'hit unpack-sources'

profile_install: Copied to *$ARTIFACT/artifact.json* before the build.

import_modify_env: Copied to *$ARTIFACT/artifact.json* before the build.

2.2.4 The build environment

See `hashdist.core.execute_job` for information about how the build job is executed. In addition, the following environment variables are set:

BUILD: Set to the build directory. This is also the starting *cwd* of each build command. This directory may be removed after the build.

ARTIFACT: The location of the final artifact. Usually this is the "install location" and should, e.g., be passed as the `--prefix` to `./configure`-style scripts.

The build specification is available under `$BUILD/build.json`, and stdout and stderr are redirected to `$BUILD/build.log`. These two files will also be present in `$ARTIFACT` after the build.

2.2.5 Discussion

Safety of the shortened IDs

HashDist will never use these to resolve build artifacts, so collision problems come in two forms:

First, automatically finding the list of run-time dependencies from the build dependencies. In this case one scans the artifact directory only for the build dependencies (less than hundred). It then makes sense to consider the chance of finding one exact string `aaa/0.0/ZXa3` in a random stream of 8-bit bytes, which helps collision strength a lot, with chance "per byte" of collision on the order $2^{-(8 \cdot 12)} = 2^{-96}$ for this minimal example.

If this is deemed a problem (the above is too optimistice), one can also scan for "duplicates" (other artifacts where longer hashes were chosen, since we know these).

The other problem can be future support for binary distribution of build artifacts, where you get pre-built artifacts which have links to other artifacts embedded, and artifacts from multiple sources may collide. In this case it makes sense to increase the hash lengths a bit since the birthday effect comes into play and since one only has 6 bits per byte. However, the downloaded builds presumably will contain the full IDs, and so on can check if there is a conflict and give an explicit error.

2.3 HashDist API reference

2.3.1 Important concepts

`hashdist.core.source_cache` — Source cache

The source cache makes sure that one doesn't have to re-download source code from the net every time one wants to rebuild. For consistency/simplicity, the software builder also requires that local sources are first "uploaded" to the cache.

The software cache currently has explicit support for tarballs, git, and storing files as-is without metadata. A "source item" (tarball, git commit, or set of files) is identified by a secure hash. The generic API in *`SourceCache.fetch()`* and *`SourceCache.unpack()`* works by using such hashes as keys. The retrieval and unpacking methods are determined by the key prefix:

```
sc.fetch('http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2',
         'tar.bz2:ttjyphyfwphjdc563imtvhnn4x4pluh5')
sc.unpack('tar.bz2:ttjyphyfwphjdc563imtvhnn4x4pluh5', '/your/location/here')

sc.fetch('https://github.com/numpy/numpy.git',
         'git:35dc14b0a59cf16be8ebdac04f7269ac455d5e43')
```

For cases where one doesn't know the key up front one uses the key-retrieving API. This is typically done in interactive settings to aid distribution/package developers:

```
key1 = sc.fetch_git('https://github.com/numpy/numpy.git', 'master')
key2 = sc.fetch_archive('http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2')
```

Features

- Re-downloading all the sources on each build gets old quickly...
- Native support for multiple retrieval mechanisms. This is important as one wants to use tarballs for slowly-changing stable code, but VCS for quickly-changing code.

- Isolates dealing with various source code retrieval mechanisms from upper layers, who can simply pass along two strings regardless of method.
- Safety: Hashes are re-checked on the fly while unpacking, to protect against corruption or tainting of the source cache.
- Should be safe for multiple users to share a source cache directory on a shared file-system as long as all have write access, though this may need some work with permissions.

Source keys

The keys for a given source item can be determined *a priori*. The rules are as follows:

Tarballs/archives: SHA-256, encoded in base64 using `format_digest()`. The prefix is currently either `tar.gz` or `tar.bz2`.

Git commits: Identified by their (SHA-1) commits prefixed with `git:`.

Individual files or directories ("hit-pack"): A tarball hash is not deterministic from the file contents alone (there's metadata, compression, etc.). In order to hash build scripts etc. with hashes based on the contents alone, we use a custom "archive format" as the basis of the hash stream. The format starts with the 8-byte magic string "HDSTPCK1", followed by each file sorted by their filename (potentially containing "/"). Each file is stored as

little-endian `uint32_t`	length of filename
little-endian `uint32_t`	length of contents
—	filename (no terminating null)
—	contents

This stream is then encoded like archives (SHA-256 in base-64), and prefixed with `files:` to get the key.

Module reference

class `hashdist.core.source_cache.`**`ProgressSpinner`**
Replacement for ProgressBar when we don't know the file length.

Methods

class `hashdist.core.source_cache.`**`SourceCache`**(*cache_path*, *logger*, *mirrors=()*, *create_dirs=False*)

Methods

static **`create_from_config`**(*config*, *logger*, *create_dirs=False*)
Creates a SourceCache from the settings in the configuration

`fetch`(**args*, ***kwargs*)
Fetch sources whose key is known.

This is the method to use in automated settings. If the sources globally identified by *key* are already present in the cache, the method returns immediately, otherwise it attempts to download the sources from *url*. How to interpret the URL is determined by the prefix of *key*.

Parameters **url** : str or None

Location to download sources from. Exact meaning depends on prefix of *key*. If *None* is passed, an exception is raised if the source object is not present.

key : str

Globally unique key for the source object.

repo_name : str or None

A unique ID for the source code repo; required for git and ignored otherwise. This must be present because a git "project" is distributed and cannot be deduced from URL (and pulling everything into the same repo was way too slow). Hopefully this can be mended in the future.

fetch_archive(*url*, *type=None*)

Fetches a tarball without knowing the key up-front.

In automated settings, `fetch()` should be used instead.

Parameters **url** : str

Where to download archive from. Local files can be specified by prepending `"file:"` to the path.

type : str (optional)

Type of archive, such as `"tar.gz"`, `"tar.gz2"`. For use when this cannot be determined from the suffix of the url.

fetch_git(*repository*, *rev*, *repo_name*)

Fetches source code from git repository

With this method one does not need to know a specific commit, but can use a generic git rev such as *master* or *revs/heads/master*. In automated settings or if the commit hash is known exactly, `fetch()` should be used instead.

Parameters **repository** : str

The repository URL (forwarded to git)

rev : str

The rev to download (forwarded to git)

repo_name : str

A unique name to use for the repository, e.g., `numpy`. This is currently required because git doesn't seem to allow getting a unique ID for a remote repo; and cloning all repos into the same git repo has scalability issues.

Returns **key** : str

The globally unique key; this is the git commit SHA-1 hash prepended by `git:`.

put(*files*)

Put in-memory contents into the source cache.

Parameters **files** : dict or list of (filename, contents)

The contents of the archive. *filename* may contain forward slashes `/` as path separators. *contents* is a pure bytes objects which will be dumped directly to *stream*.

Returns **key** : str

The resulting key, it has the `files:` prefix.

unpack(*key*, *target_path*)

Unpacks the sources identified by *key* to *target_path*

The sources are verified against their secure hash to guard against corruption/security problems. *CorruptSourceCacheError* will be raised in this case. In normal circumstances this should never happen.

The archive will be loaded into memory, checked against the hash, and then extracted from the memory copy, so that attacks through tampering with on-disk archives should not be possible.

Parameters **key** : str

The source item key/secure hash

target_path : str

Path to extract in

class hashdist.core.source_cache.**TarSubprocessHandler**(*logger*)

Call external tar

This handler should only be used as fallback, it lacks some features and/or depends on the vagueries of the host tar.

Methods

hashdist.core.source_cache.**hit_pack**(*files*, *stream=None*)

Packs the given files in the "hit-pack" format documented above, and returns the resulting key. This is useful to hash a set of files solely by their contents, not metadata, except the filename.

Parameters **files** : list of (filename, contents)

The contents of the archive. *filename* may contain forward slashes `/` as path separators. *contents* is a pure bytes objects which will be dumped directly to *stream*.

stream : file-like (optional)

Result of the packing, or *None* if one only wishes to know the hash.

Returns **The key of the resulting pack** :

(e.g., ''files:cmRX4RyxU63D9Ciq8ZAfxWGjdMMOXn2mdCwHQqM4Zjw''). :

hashdist.core.source_cache.**hit_unpack**(*stream*, *key*)

Unpacks the files in the "hit-pack" format documented above, verifies that it matches the given key, and returns the contents (in memory).

Parameters **stream** : file-like

Stream to read the pack from

key : str

Result from *hit_pack()*.

Returns **list of (filename, contents)** :

hashdist.core.source_cache.**scatter_files**(*files*, *target_dir*)

Given a list of filenames and their contents, write them to the file system.

Will not overwrite files (raises an OSError(errno.EEXIST)).

This is typically used together with *hit_unpack()*.

Parameters **files** : list of (filename, contents)

target_dir : str

Filesystem location to emit the files to

`hashdist.core.build_store` — Build artifact store

Principles

The build store is the very core of HashDist: Producing build artifacts identified by hash-IDs. It's important to have a clear picture of just what the build store is responsible for and not.

Nix takes a pure approach where an artifact hash is guaranteed to identify the resulting binaries (up to anything inherently random in the build process, like garbage left by compilers). In contrast, HashDist takes a much more lenient approach where the strictness is configurable. The primary goal of HashDist is to make life simpler by reliably triggering rebuilds when software components are updated, not total control of the environment (in which case Nix is likely the better option).

The *only* concern of the build store is managing the result of a build. So the declared dependencies in the build-spec are not the same as "package dependencies" found in higher-level distributions; for instance, if a pure Python package has a NumPy dependency, this should not be declared in the build-spec because NumPy is not needed during the build; indeed, the installation can happen in parallel.

Artifact IDs

A HashDist artifact ID has the form `name/hash`, e.g., `zlib/4niostz3iktlg67najtxuwwgss5vl6k4`.

For the artifact paths on disk, a shortened form (4-char hash) is used to make things more friendly to the human user. If there is a collision, the length is simply increased for the one that comes later. Thus, the example above could be stored on disk as `~/.hit/opt/zlib/4nio`, or `~/.hit/opt/zlib/1.2.7/4nios` in the (rather unlikely) case of a collision. There is a symlink from the full ID to the shortened form. See also Discussion below.

Build specifications and inferring artifact IDs

The fundamental object of the build store is the JSON build specification. If you know the build spec, you know the artifact ID, since the latter is the hash of the former. The key is that both *dependencies* and *sources* are specified in terms of their hashes.

An example build spec:

```
{
    "name" : "<name of piece of software>",
    "description": "<what makes this build special>",
    "build": {
        "import" : [
             {"ref": "bash", "id": "virtual:bash"},
             {"ref": "make", "id": "virtual:gnu-make/3+"},
             {"ref": "zlib", "id": "zlib/1.2.7/fXHu+8dcqmREfXaz+ixMkh2LQbvIKlHf+rtl5HEfgmU"},
             {"ref": "unix", "id": "virtual:unix"},
             {"ref": "gcc", "id": "gcc/host-4.6.3/q0VSL7JmzH1P17meqITYc4kMbnIjIexrWPdlAlqPn3s", "bef
         ],
         "commands" : [
             {"cmd": ["bash", "build.sh"]}
         ],
     },
     "sources" : [
         {"key": "git:c5ccca92c5f136833ad85614feb2aa4f5bd8b7c3"},
         {"key": "tar.bz2:RB1JbykVljxdvL07mN60y9V9BVCruWRky2FpK2QCCow", "target": "sources"},
         {"key": "files:5fcANXHsmjPpukSffBZF913JEnMwzcCoysn-RZEX7cM"}
     ],
}
```

name: Should match `[a-zA-Z0-9-_+]+`.

version: Should match `[a-zA-Z0-9-_+]*`.

build: A job to run to perform the build. See *hashdist.core.run_job* for the documentation of this sub-document.

sources: Sources are unpacked; documentation for now in 'hit unpack-sources'

The build environment

See `hashdist.core.execute_job` for information about how the build job is executed. In addition, the following environment variables are set:

BUILD: Set to the build directory. This is also the starting *cwd* of each build command. This directory may be removed after the build.

ARTIFACT: The location of the final artifact. Usually this is the "install location" and should, e.g., be passed as the `--prefix` to `./configure`-style scripts.

The build specification is available under `$BUILD/build.json`, and stdout and stderr are redirected to `$BUILD/_hashdist/build.log`. These two files will also be present in `$ARTIFACT` after the build.

Build artifact storage format

The presence of the 'id' file signals that the build is complete, and contains the full 256-bit hash.

More TODO.

Reference

class hashdist.core.build_store.**BuildSpec**(*build_spec*)

Wraps the document corresponding to a build.json

The document is wrapped in order to a) signal that is has been canonicalized, b) make the artifact id available under the artifact_id attribute.

class hashdist.core.build_store.**BuildStore**(*temp_build_dir*, *artifact_root*, *gc_roots_dir*, *logger*, *create_dirs=False*)

Manages the directory of build artifacts; this is usually the entry point for kicking off builds as well.

Parameters **temp_build_dir** : str

Directory to use for temporary builds (these may be removed or linger depending on *keep_build* passed to *ensure_present()*).

artifact_root : str

Root of artifacts, this will be prepended to artifact_path_pattern with *os.path.join*. While this could be part of *artifact_path_pattern*, the meaning is that garbage collection will never remove contents outside of this directory.

gc_roots_dir : str

Directory of symlinks to symlinks to artifacts. Artifacts reached through these will not be collected in garbage collection.

logger : Logger

Methods

static **create_from_config**(*config*, *logger*, ***kw*)
Creates a SourceCache from the settings in the configuration

create_symlink_to_artifact(*artifact_id*, *symlink_target*)
Creates a symlink to an artifact (usually a 'profile')

The symlink can be placed anywhere the users wants to access it. In addition to the symlink being created, it is listed in gc_roots.

The symlink will be created atomically, any target file/symlink will be overwritten.

delete(*artifact_id*)
Deletes an artifact ID from the store. This is simply an *rmtree*, i.e., it is (at least currently) possible to delete an aborted build, a build in progress etc., as long as it is present in the right path.

This is the backend of the `hit purge` command.

Returns the path that was removed, or *None* if no path was present.

ensure_present(*build_spec*, *config*, *extra_env=None*, *virtuals=None*, *keep_build='never'*, *debug=False*)
Builds an artifact (if it is not already present).

extra_env: dict (optional) Extra environment variables to pass to the build environment. These are *NOT* hashed!

gc()
Run garbage collection, removing any unneeded artifacts.

For now, this doesn't care about virtual dependencies. They're not used at the moment of writing this; it would have to be revisited in the future.

make_artifact_dir(*build_spec*)
Makes a directory to put the result of the artifact build in. This does not register the artifact in the db (which should be done after the artifact is complete).

make_build_dir(*build_spec*)
Creates a temporary build directory

Just to get a nicer name than mkdtemp would. The caller is responsible for removal.

resolve(*artifact_id*)
Given an artifact_id, resolve the short path for it, or return None if the artifact isn't built.

hashdist.core.build_store.**assert_safe_name**(*x*)
Raises a ValueError if x does not match `[a-zA-Z0-9-_+]+`.

Returns *x*

hashdist.core.build_store.**canonicalize_build_spec**(*spec*)
Puts the build spec on a canonical form + basic validation

See module documentation for information on the build specification.

Parameters **spec** : json-like

The build specification

Returns **canonical_spec** : json-like

Canonicalized and verified build spec

hashdist.core.build_store.**shorten_artifact_id**(*artifact_id*, *length=12*)
Shortens the hash part of the artifact_id to the desired length

hashdist.core.build_store.**strip_comments**(*spec*)
Strips a build spec (which should be in canonical format) of comments that should not affect hash

hashdist.core.build_store.**unpack_sources**(*logger*, *source_cache*, *doc*, *target_dir*)
Executes source unpacking from 'sources' section in build.json

hashdist.core.run_job — Job execution in controlled environment

Executes a set of commands in a controlled environment, determined by a JSON job specification. This is used as the "build" section of `build.json`, the "install" section of `artifact.json`, and so on.

The job spec may not completely specify the job environment because it is usually a building block of other specs which may imply certain additional environment variables. E.g., during a build, `$ARTIFACT` and `$BUILD` are defined even if they are never mentioned here.

Job specification

The job spec is a document that contains what's needed to set up a controlled environment and run the commands. The idea is to be able to reproduce a job run, and hash the job spec. Example:

```
{
    "import" : [
        {"ref": "BASH", "id": "virtual:bash"},
        {"ref": "MAKE", "id": "virtual:gnu-make/3+"},
        {"ref": "ZLIB", "id": "zlib/2d4kh7hw4uvml67q7npltyaau5xmn4pc"},
        {"ref": "UNIX", "id": "virtual:unix"},
        {"ref": "GCC", "id": "gcc/jonykztnjeqm7bxurpjuttsprphbooqt"}
     ],
     "commands" : [
         {"chdir": "src"},
         {"prepend_path": "FOOPATH", "value": "$ARTIFACT/bin"},
         {"set": "INCLUDE_FROB", "value": "0"},
         {"cmd": ["pkg-config", "--cflags", "foo"], "to_var": "CFLAGS"},
         {"cmd": ["./configure", "--prefix=$ARTIFACT", "--foo-setting=$FOO"]}
         {"cmd": ["bash", "$in0"],
          "inputs": [
              {"text": [
                  "["$RUN_FOO" != "" ] && ./foo"
                  "make",
                  "make install"
              ]}
         }
     ],
}
```

Job spec root node

The root node is also a command node, as described below, but has two extra allowed keys:

import: The artifacts needed in the environment for the run. After the job has run they have no effect (i.e., they do not affect garbage collection or run-time dependencies of a build, for instance). The list is ordered and earlier entries are imported before latter ones.

- **id**: The artifact ID. If the value is prepended with `"virtual:"`, the ID is a virtual ID, used so that the real one does not contribute to the hash. See section on virtual imports below.
- **ref**: A name to use to inject information of this dependency into the environment. Above, `$ZLIB_DIR` will be the absolute path to the `zlib` artifact, and `$ZLIB_ID` will be the full artifact ID. This can be set to *None* in order to not set any environment variables for the artifact.

When executing, the environment is set up as follows:

- Environment is cleared (`os.environ` has no effect)
- The initial environment provided by caller (e.g., *BuildStore* provides *$ARTIFACT* and *$BUILD*) is loaded
- The *import* section is processed
- Commands executed (which may modify env)

Command node

The command nodes is essentially a script language, but lacks any form of control flow. The purpose is to control the environment, and then quickly dispatch to a script in a real programming language.

Also, the overall flow of commands to set up the build environment are typically generated by a pipeline from a package definition, and generating a text script in a pipeline is no fun.

See example above for basic script structure. Rules:

- Every item in the job is either a *cmd* or a *commands* or a *hit*, i.e. those keys are mutually exclusive and defines the node type.
- *commands*: Push a new environment and current directory to stack, execute sub-commands, and pop the stack.
- *cmd*: The list is passed straight to `subprocess.Popen()` as is (after variable substitution). I.e., no quoting, no globbing.
- *hit*: executes the *hit* tool *in-process*. It acts like *cmd* otherwise, e.g., *to_var* works.
- *chdir*: Change current directory, relative to current one (same as modifying *PWD* environment variable)
- *set*, *prepend/append_path*, *prepend/append_flag*: Change environment variables, inserting the value specified by the *value* key, using variable substitution as explained below. *set* simply overwrites variable, while the others modify path/flag-style variables, using the *os.path.patsep* for *prepend/append_path* and a space for *prepend/append_flag*. **NOTE:** One can use *nohash_value* instead of *value* to avoid the value to enter the hash of a build specification.
- *files* specifies files that are dumped to temporary files and made available as *$in0*, *$in1* and so on. Each file has the form `{typestr: value}`, where *typestr* means:
  - `text`: *value* should be a list of strings which are joined by newlines
  - `string`: *value* is dumped verbatim to file
  - `json`: *value* is any JSON document, which is serialized to the file
- stdout and stderr will be logged, except if *to_var* or *append_to_file* is present in which case the stdout is capture to an environment variable or redirected in append-mode to file, respectively. (In the former case, the resulting string undergoes *strip()*, and is then available for the following commands within the same scope.)
- Variable substitution is performed the following places: The *cmd*, *value* of *set* etc., *chdir* argument, *stdout_to_file*. The syntax is `$CFLAGS` and `${CFLAGS}`. `\$` is an escape for `$`, `\` is an escape for `\`, other escapes not currently supported and `\` will carry through unmodified.

For the *hit* tool, in addition to what is listed in `hit --help`, the following special command is available for interacting with the job runner:

- `hit logpipe HEADING LEVEL`: Creates a new Unix FIFO and prints its name to standard output (it will be removed once the job terminates). The job runner will poll the pipe and print anything written to it nicely formatted to the log with the given heading and log level (the latter is one of `DEBUG`, `INFO`, `WARNING`, `ERROR`).

Note: `hit` is not automatically available in the environment in general (in launched scripts etc.), for that, see `hashdist.core.hit_recipe`. `hit logpipe` is currently not supported outside of the job spec at all (this could be supported through RPC with the job runner, but the gain seems very slight).

Virtual imports

Some times it is not desirable for some imports to become part of the hash. For instance, if the `cp` tool is used in the job, one is normally ready to trust that the result wouldn't have been different if a newer version of the `cp` tool was used instead.

Virtual imports, such as `virtual:unix` in the example above, are used so that the hash depends on a user-defined string rather than the artifact contents. If a bug in `cp` is indeed discovered, one can change the user-defined string (e.g, `virtual:unix/r2`) in order to change the hash of the job desc.

Note: One should think about virtual dependencies merely as a tool that gives the user control (and responsibility) over when the hash should change. They are *not* the primary mechanism for providing software from the host; though software from the host will sometimes be specified as virtual dependencies.

Reference

class hashdist.core.run_job.**CommandTreeExecution**(*logger*, *temp_dir=None*, *debug=False*, *debug_shell='/bin/bash'*)

Class for maintaining state (in particular logging pipes) while executing script. Note that the environment is passed around as parameters instead.

Executing `run()` multiple times amounts to executing different variable scopes (but with same logging pipes set up).

Parameters **logger** : Logger

rpc_dir : str

A temporary directory on a local filesystem. Currently used for creating pipes with the "hit logpipe" command.

Methods

close()

Removes log FIFOs; should always be called when one is done

dump_inputs(*inputs*, *node_pos*)

Handles the 'inputs' attribute of a node by dumping to temporary files.

Returns **A dict with environment variables that can be used to update 'env',** :

containing ''$in0'', ... :

logged_check_call(*args*, *env*, *stdout_to*)
Similar to subprocess.check_call, but multiplexes input from stderr, stdout and any number of log FIFO pipes available to the called process into a single Logger instance. Optionally captures stdout instead of logging it.

run_hit(*args*, *env*, *stdout_to=None*)
Run `hit` in the same process.

But do not emit INFO-messages from sub-command unless level is DEBUG.

run_node(*node*, *env*, *node_pos*)
Executes a script node and its children

Parameters **node** : dict

A command node

env : dict

The environment (will be modified). The PWD variable tracks working directory and should always be set on input.

node_pos : tuple

Tuple of the "path" to this command node; e.g., (0, 1) for second command in first group.

hashdist.core.run_job.**canonicalize_job_spec**(*job_spec*)
Returns a copy of job_spec with default values filled in.

Also performs a tiny bit of validation.

hashdist.core.run_job.**handle_imports**(*logger*, *build_store*, *artifact_dir*, *virtuals*, *job_spec*)
Sets up environment variables for a job. This includes $MYIMPORT_DIR, $MYIMPORT_ID, $ARTIFACT, $HDIST_IMPORT, $HDIST_IMPORT_PATHS.

Returns **env** : dict

Environment containing HDIST_IMPORT{,_PATHS} and variables for each import.

script : list

Instructions to execute; imports first and the job_spec commands afterwards.

hashdist.core.run_job.**run_job**(*logger*, *build_store*, *job_spec*, *override_env*, *artifact_dir*, *virtuals*, *cwd*, *config*, *temp_dir=None*, *debug=False*)
Runs a job in a controlled environment, according to rules documented above.

Parameters **logger** : Logger

build_store : BuildStore

BuildStore to find referenced artifacts in.

job_spec : document

See above

override_env : dict

Extra environment variables not present in job_spec, these will be added last and overwrite existing ones.

artifact_dir : str

The value $ARTIFACT should take after running the imports

virtuals : dict

Maps virtual artifact to real artifact IDs.

cwd : str

The starting working directory of the job. Currently this cannot be changed (though a `cd` command may be implemented in the future if necesarry)

config : dict

Configuration from `hashdist.core.config`. This will be serialied and put into the HDIST_CONFIG environment variable for use by `hit`.

temp_dir : str (optional)

A temporary directory for use by the job runner. Files will be left in the dir after execution.

debug : bool

Whether to run in debug mode.

Returns out_env: dict :

The environment after the last command that was run (regardless of scoping/nesting). If the job spec is empty (no commands), this will be an empty dict.

`hashdist.core.run_job.`**`substitute`**(*x*, *env*)

Substitute environment variable into a string following the rules documented above.

Raises KeyError if an unreferenced variable is not present in env (`$$` always raises KeyError)

`hashdist.spec.profile` — HashDist Profiles

Not supported:

- Diamond inheritance

class `hashdist.spec.profile.`**`FileResolver`**(*checkouts_manager*, *search_dirs*)

Find spec files in an overlay-based filesystem, consulting many search paths in order. Supports the `<repo_name>/some/path`-convention.

Methods

`find_file`(*filenames*)

Search for a file.

Search for a file with the given filename/path relative to the root of each of `self.search_dirs`.

Parameters filenames : list of strings

Filenames to seach for. The entire list will be searched before moving on to the next layer/overlay.

Returns Returns the found file (in the :

‘‘<repo_name>/some/path‘‘-convention), or None if no file was :

found. :

`glob_files`(*patterns*, *match_basename=False*)

Match file globs.

Like `find_file`, but uses a set of patterns and tries to match each pattern against the filesystem using `glob.glob`.

Parameters **patterns** : list of strings

Glob patterns

match_basename : boolean

If `match_basename` is set, only the basename of the file is compared (i.e., one file with each basename will be returned).

Returns **The result is a dict mapping the "matched name" to a pair** :

(pattern, full qualified path). :

*** The matched name is a path relative to root of overlay** :

(required to be unique) or just the basename, depending on `match_basename`.

*** The pattern returned is the pattern that whose match gave** :

rise to the "matched path" key.

*** The full qualified name is the filename that was mattched by** :

the pattern.

class `hashdist.spec.profile.`**`PackageYAML`**(*used_name*, *filename*, *parameters*, *in_directory*)

Holds content of a package yaml file

The content is unmodified except for *{{VAR}}* variable expansion.

Attributes

doc	dict	The deserialized yaml source
in_directory	boolean	Whether the yaml file is in its private package directory that may contain other files.
parameters	dict of str	Parameters with the defaults from the package yaml file applied
filename	str	Full qualified name of the package yaml file
hook_filename	str or `None`	Full qualified name of the package `.py` hook file, if it exists.

`dirname`

Name of the package directory.

Returns **String, full qualified name of the directory containing the** :

yaml file. :

Raises **A ''ValueError'' is raised if the package is a stand-alone** :

yaml file, that is, there is no package directory. :

class `hashdist.spec.profile.`**`Profile`**(*logger*, *doc*, *checkouts_manager*)

Profiles acts as nodes in a tree, with *extends* containing the parent profiles (which are child nodes in a DAG).

Methods

`find_package_file`(*pkgname*, *filename*)

Find a package resource file.

Search for the file at:

- `$pkgs/filename`,
- `$pkgs/pkgname/filename`,

in this order.

Parameters **pkgname** : string

Name of the package (excluding `.yaml`).

filename : string

File name to look for.

Returns **The full qualifiedfilename as a string, or ‘‘None‘‘ if no file** :

is found. :

load_package_yaml (*pkgname*, *parameters*)

Search for the yaml source and load it.

Load the source for `pkgname` (after substitution by a `use:` profile section, if any) from either

- `$pkgs/pkgname.yaml`,
- `$pkgs/pkgname/pkgname.yaml`, or
- `$pkgs/pkgname/pkgname-*.yaml`

by searching through the paths in the `package_dirs:` profile setting. The paths are searched order, and only the first match per basename is used. That is, `$pkgs/foo/foo.yaml` overrides `$pkgs/foo.yaml`. And `$pkgs/foo/foo-bar.yaml` is returned in addition.

In case of many matches, any when'-clause is evaluated on *parameters* and a single document should result, otherwise an exception is raised. A document without a when-clause is overridden by those with a when-clause.

Parameters **pkgname** : string

Name of the package (excluding `.yaml`).

parameters : dict

The profile parameters.

Returns **A :class:‘PackageYAML‘ instance if successfull.** :

Raises *** class:‘~hashdist.spec.exceptions.ProfileError‘‘ is raised if** :

there is no such package.

*** class:‘~hashdist.spec.exceptions.PackageError‘‘ is raised if** :

a package conflicts with a previous one.

resolve (*path*)

Turn <repo>/path into /tmp/foo-342/path

class `hashdist.spec.profile.`**TemporarySourceCheckouts** (*source_cache*)

A context that holds a number of sources checked out to temporary directories until it is released.

Methods

resolve(*path*)
Expand path-names of the form `<repo_name>/foo/bar`, replacing the `<repo_name>` part (where `repo_name` is given to *checkout*, and the < and > are literals) with the temporary checkout of the given directory.

hashdist.spec.profile.**load_and_inherit_profile**(*checkouts*, *include_doc*, *cwd=None*, *override_parameters=None*)
Loads a Profile given an include document fragment, e.g.:

```
file: ../foo/profile.yaml
```

or:

```
file: linux/profile.yaml
urls: [git://github.com/hashdist/hashstack.git]
key: git:5aeba2c06ed1458ae4dc5d2c56bcf7092827347e
```

The load happens recursively, including fetching any remote dependencies, and merging the result into this document.

cwd is where to interpret *file* in *include_doc* relative to (if it is not in a temporary checked out source). It can use the format of TemporarySourceCheckouts, `<repo_name>/some/path`.

2.3.2 Support code

hashdist.core.hasher – Utilities for hashing

class hashdist.core.hasher.**DocumentSerializer**(*wrapped*)
Stable non-Python-specific serialization of nested objects/documents. The primary usecase is for hashing (see *Hasher*), and specifically hashing of JSON documents, thus no de-serialization is implemented. The idea is simply that by hashing a proper serialization format we ensure that we don't weaken the hash function.

The API used is that of `hashlib` (i.e. an update method).

A core goal is that it should be completely stable, and easy to reimplement in other languages. Thus we stay away from Python-specific pickling mechanisms etc.

Supported types: Basic scalars (ints, floats, True, False, None), bytes, unicode, and buffers, lists/tuples and dicts.

Additionally, when encountering user-defined objects with the `get_secure_hash` method, that method is called and the result used as the "serialization". The method should return a tuple (type_id, secure_hash); the former should be a string representing the "type" of the object (often the fully qualified class name), in order to avoid conflicts with the hashes of other objects, and the latter a hash of the contents.

The serialization is "type-safe" so that `"3"` and `3` and `3.0` will serialize differently. Lists and tuples are treated as the same (`(1,)` and `[1]` are the same) and buffers, strings and Unicode objects (in their UTF-8 encoding) are also treated the same.

Note: Currently only string keys are supported for dicts, and the items are serialized in the order of the keys. This is because all Python objects implement comparison, and comparing by arbitrary Python objects could lead to easy misuse (hashes that are not stable across processes).

One could instead sort the keys by their hash (getting rid of comparison), but that would make the hash-stream (and thus the unit tests) much more complicated, and the idea is this should be reproducible in other languages. However that is a possibility for further extension, as long as string keys are treated as today.

In order to prevent somebody from constructing colliding documents, each object is hashed with an envelope specifying the type and the length (in number of items in the case of a container, or number of bytes in the case of str/unicode/buffer).

In general, see unit tests for format examples/details.

Parameters **wrapped** : object

wrapped.update is called with strings or buffers to emit the resulting stream (the API of the `hashlib` hashers)

Methods

class `hashdist.core.hasher.`**Hasher**(*x=None*)

Cryptographically hashes buffers or nested objects ("JSON-like" object structures). See *DocumentSerializer* for more details.

This is the standard hashing method of HashDist.

Methods

format_digest()

The HashDist standard digest.

class `hashdist.core.hasher.`**HashingReadStream**(*hasher*, *stream*)

Utility for reading from a stream and hashing at the same time.

Methods

class `hashdist.core.hasher.`**HashingWriteStream**(*hasher*, *stream*)

Utility for hashing and writing to a stream at the same time. The *stream* may be *None* for convenience.

Methods

`hashdist.core.hasher.`**check_no_floating_point**(*doc*)

Verifies that the document *doc* does not contain floating-point numbers.

`hashdist.core.hasher.`**format_digest**(*hasher*)

The HashDist standard format for encoding hash digests

This is one of the cases where it is prudent to just repeat the implementation in the docstring:

```
base64.b32encode(hasher.digest()[:20]).lower()
```

Parameters **hasher** : hasher object

An object with a *digest* method (a *Hasher* or an object from the `hashlib` module)

`hashdist.core.hasher.`**hash_document**(*doctype*, *doc*)

Computes a hash from a document. This is done by serializing to as compact JSON as possible with sorted keys, then perform sha256 an. The string `{doctype}|` is prepended to the hashed string and serves to make sure different kind of documents yield different hashes even if they are identical.

Some unicode characters have multiple possible code-points, so that this definition; however, this should be considered an extreme corner case. In general it should be very unusual for hashes that are publicly shared/moves

beyond one computer to contain anything but ASCII. However, we do not enforce this, in case one wishes to encode references in the local filesystem.

Floating-point numbers are not supported (these have multiple representations).

`hashdist.core.hasher.`**`prune_nohash`**(*doc*)
Returns a copy of the document with every key/value-pair whose key starts with `'nohash_'` is removed.

`hashdist.core.links` — Link creation tool

`execute_links_dsl()` takes a set of rules in a mini-language and uses it to create (a potentially large number of) links.

The following rules creates links to everything in "/usr/bin", except for "/usr/bin/gcc-4.6" which is copied (though it could be achieved more easily with the *overwrite* flag):

```
[
  {
    "action": "copy",
    "source": "/usr/bin/gcc-4.6",
    "target": "$ARTIFACT/bin/gcc-4.6"
  },
  {
    "action": "exclude",
    "select": ["/usr/bin/gcc-4.6", "/something/else/too/**"]
  },
  {
    "action": "symlink",
    "select": "/usr/bin/*",
    "prefix": "/usr",
    "target": "$ARTIFACT"
  }

]
```

Rules are executed in order. If a target file already exists, nothing happens.

action: One of "symlink", "absolute_symlink", "relative_symlink", "copy", "exclude", "launcher". Other types may be added later.

- *absolute_symlink* creates absolute symlinks. Just *symlink* is an alias for absolute symlink.
- *relative_symlink* creates relative symlinks
- *copy* copies contents and mode (`shutil.copy`)
- *exclude* makes sure matching files are not considered in rules below
- *launcher*, see *`make_launcher()`*

select, prefix: *select* contains glob of files to link/copy/exclude. This is in ant-glob format (see *`hashdist.core.ant_glob`*). If *select* is given and *action* is not *exclude*, one must also supply a *prefix* (possibly empty string) which will be stripped from each matching path, before recreating the same hierarchy beneath *target*.

Variable substitution is performed both in *select* and *prefix*. *select* and *prefix* should either both be absolute paths or both be relative paths

source: Provide an exact filename instead of a glob. In this case *target* should refer to the exact filename of the resulting link/copy.

target: Target filename (*source* is used) or directory (*select* is used). Variable substitution is performed.

dirs: If present and *True*, symlink matching directories, not only files. Only takes effect for *select*; *source* always selectes dirs.

overwrite: If present and *True*, overwrite target.

`hashdist.core.links.`**`dry_run_links_dsl`**(*rules*, *env={}*)

Turns a DSL for creating links/copying files into a list of actions to be taken.

This takes into account filesystem contents and current directory at the time of call.

See *execute_links_dsl()* for information on the DSL.

Parameters **rules** : list

List of rules as described in *execute_links_dsl()*.

env : dict

Environment to use for variable substitution

Returns **actions** : list

What actions should be performed as a list of (func,) + args_to_pass, where *func* is one of *os.symlink*, `silent_makedirs()`, *shutil.copyfile*.

`hashdist.core.links.`**`execute_links_dsl`**(*rules*, *env={}*, *launcher_program=None*, *logger=None*)

Executes the links DSL for linking/copying files

The input is a set of rules which will be applied in order. The rules are documented above.

Parameters **spec** : list

List of rules as described above.

env : dict

Environment to use for variable substitution.

launcher_program : str or None

If the 'launcher' action is used, the path to the launcher executable must be provided.

logger : Logger or `None` (default)

`hashdist.core.links.`**`make_launcher`**(*src*, *dst*, *launcher_program*)

The 'launcher' action. This is a general tool for processing bin-style directories where the binaries must have a "fake" argv[0] target (read: Python). The action depends on the source type:

program (i.e., executable not starting with #!): Set up as symlink to "launcher", which is copied into same directory; and "$dst.link" is set up to point relatively to "$src".

symlink: Copy it verbatim. Thus, e.g., `python -> python2.7` will point to the newly, "launched-ified" `python2.7`.

other (incl. scripts): Symlink relatively to it.

`hashdist.core.ant_glob` – ant-inspired globbing

`hashdist.core.ant_glob.`**`ant_iglob`**(*pattern*, *cwd=''*, *include_dirs=True*)

Generator that iterates over files/directories matching the pattern.

The syntax is ant-glob-inspired but currently only a small subset is implemented.

Examples:

```
    *.txt         # matches "a.txt", "b.txt"
    foo/**/bar    # matches "foo/bar" and "foo/a/b/c/bar"
    foo*/**/*.bin # matches "foo/bar.bin", "foo/a/b/c/bar.bin", "foo3/a.bin"
```

Illegal patterns:

```
    foo/**.bin  # '**' can only match 0 or more entire directories
```

Parameters **pattern** : str or list

Glob pattern as described above. If a str, will be split by /; if a list, each item is a path component. It is only possible to specify a non-relative glob if *pattern* is a string.

cwd : str

Directory to start in. Pass an empty string or '.' for current directory; the former will emit 'rel/path/to/file' while the latter './rel/path/to/file'. (This added complexity is present in order to be able to reliably match prefixes by string value).

include_dirs : bool

Whether to include directories, or only glob files.

`hashdist.core.ant_glob.`**`has_permission`**(*path*)

Returns True if we have 'listdir' permissions. False otherwise.

2.4 Plans for HashDist

2.4.1 HashDist core components

Source store

The idea of the source store is to help download source code from the net, or "upload" files from the local disk. Items in the store are identified with a cryptographic hash.

Items in the source store are only needed for a few hours while the build takes place, and that may be the default configuration for desktop users. However, an advantage of keeping things around forever is to always be able to redo an old build without relying on third parties. Also, if many users share a source store on the local network one can reduce the average time spent waiting for downloads. This is an aspect that will be very different for different userbases.

Here's how to fetch some sources; the last line output (only one to `stdout`) is the resulting key:

```
$ hit fetch http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2
Downloading... progress indicator ...
Done
targz:mheIiqyFVX61qnGc53ZhR-uqVsY

$ hit fetch git://github.com/numpy/numpy.git master
Fetching ...
Done
git:c5ccca92c5f136833ad85614feb2aa4f5bd8b7c3
```

One can then unpack results later only by using the key:

```
$ hit unpack targz:mheIiqyFVX61qnGc53ZhR-uqVsY src/python
$ hit unpack git:c5ccca92c5f136833ad85614feb2aa4f5bd8b7c3 numpy
```

While `targz:` and `git:` is part of the key, this is simply to indicate (mostly to humans) where the sources originally came from, and not a requirement of the underlying source store implementation.

Note that git source trees are simply identified by their git commits, not by an additional "repository name" or similar (the simplest implementation of this is to pull all git objects into the same local repository).

Re-fetching sources that are in cache already are not downloaded and results in the same hash:

```
$ hit fetch http://python.org/ftp/python/2.7.3/Python-2.7.3.tar.bz2
targz:mheIiqyFVX61qnGc53ZhR-uqVsY
```

It's assumed that the content under the URL will not change (at least by default). Downloading the same content from a different URL leads to de-duplication and the registration of an additional URL for that source.

Finally it's possible to store files from the local filesystem:

```
$ hit fetch /home/dagss/code/fooproject
dir:lkQYr_eQ13Sra5EYoXTg3c8msXs
$ hit fetch -ttransient /home/dagss/code/buildfoo.sh
file:tMwPj0cxhQVsA1pncZKcwCMgVbU
```

This simply copies files from local drive (mainly to make sure a copy is preserved in pristine condition for inspection if a build fails).

Tags: The `-ttransient` option was used above to tag the `buildfoo.sh` script, meaning it's not likely to be important (or is archived by other systems on local disk) so just keep it for a few days. In general we have a system of arbitrary tags which one can then make use of when configuring the GC.

Artifact builder

Assume in `~/.hashdistrc`:

```
[builder]
    store = ~/.hashdist/artifacts
```

The builder is responsible for executing builds, under the following conditions:

- The builder will *not* recurse to build a dependency. All software dependencies are assumed to have been built already (or be present on the host system).
- All sources, scripts etc. has been downloaded/uploaded to the source store

Invoking a build:

```
$ hit build < buildspec.json
Not present in store, need to build. Follow log with

    tail -f /home/dagss/.hashdist/artifacts/numpy/2.6/_build0/build.log

Done building, artifact name:
numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo

$ hit resolve numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo
/home/dagss/.hashdist/artifacts/numpy/2.6/Ymm0C_HRoH0HxNM9snC3lvcIkMo
```

The build specification may look like this for a build:

```
{
    "name" : "numpy",
    "version" : "1.6",
    "build-dependencies" : {
```

```
        "blas" : "ATLAS-3.10.0-gijMQibuq39SCBQfy5XoBeMSQKw",
        "gcc" : "gcc-4.6.3-A8x1ZV5ryXvVGLUwoeP2C01LtsY",
        "python" : "python-2.7-io-lizHjC4h8z5e2Q00Ag9xUvus",
        "bash" : "python-4.2.24.1-Z8GcCVzYOOH97n-ZC6qhfQhciCI"
    },
    "sources" : {
        "numpy" : "git:c5ccca92c5f136833ad85614feb2aa4f5bd8b7c3",
        "build.sh" : "file:gijMQibuq39SCBQfy5XoBeMSQKw",
    },
    "command" : ["$bash/bin/bash", "build.sh"],
    "env" : {
        "NUMPYLAPACKTYPE" : "ATLAS"
    },
    "env_nohash" : {
        "MAKEFLAGS" : "-j4",
    },
    "parameters" : [
        "this is free-form json", "build script can parse this information",
        "and use it as it wants"
    ],
    "parameters_nohash" : {
        "again-we-have" : ["custom", "json"]
    }
}
```

What happens:

1. A hash is computed of the contents of the build specification. This is simple since all dependencies are given in terms of their hash. Then, look up in the store; if found, we are done. (Dictionaries are supposed to be unordered and sorted prior to hashing.)
2. Let's assume the artifact doesn't exist. A temporary directory is created for the build using `mkdtemp` (this is important so that there's no races if two people share the store and attempt the same build at the same time; the directory is moved atomically to its final location after the build).
3. `chdir` to that directory, redirect all output to `build.log`, and store the build spec as `build.json`. Unpack the sources listed using the equivalent of `hit unpack`. The result in this case is a `numpy` subdirectory with the git checkout, and a `build.sh` script.
4. Set environment variables (as documented elsewhere, TBD). The keys in the *build-dependencies* section maps to environment variable names, so that `$blas` will contain `ATLAS-3.10.0-gijMQibuq39SCBQfy5XoBeMSQKw` and `$blaspath` will contain `../../ATLAS/3.10.0/gijMQibuq39SCBQfy5XoBeMSQKw`. This is the sole purpose of the keys in the *build-dependencies* section. (Build scripts may also choose to parse `build.json` too instead of relying on the environment.).
5. Set up a sandbox environment. The sandboxing should be the topic of another section.
6. Execute the given command. The command **must** start with a variable substitution of one of the dependencies listed, unless it is `hit`. (The bootstrapping problem this creates should be treated in another section.)

Build policy

It's impossible to control everything, and one needs to trust the builds that are being run that they will produce the same output given the same input. The `hit build` tool is supposed to be a useful part in bigger stack, and that bigger stack is what needs to make the tradeoffs between fidelity and practicality.

One example of this is the `X_nohash` options, which provide for passing options that only controls *how* things are

built, not *what*, so that two builds with different such entries will have the same artifact hash in the end. The builder neither encourages nor discourages the use of these options; that decision can only be made by the larger system considering a specific userbase.

Build environment and helpers

A set of conventions and utilities are present to help build scripts.

Dependency injection

Builds should never reach out and detect settings or software (except for very special bootstrapping packages), they should always get the artifacts of all dependencies injected through `build.json`.

This is largely something we cannot enforce and where one relies on sane use of the system.

(Nix builds its own toolchain in order to strictly enforce this, we consider that a too high cost to pay.)

Temporary build profiles

If one calls `hit makebuildprofile build.json`, then `build.json` is parsed and a profile environment created containing all build dependencies, whose path is then printed to standard output. Thus one can hand a single set of paths to ones scripts rather than one path per dependency.

This isn't necesarrily a recommended mode of working, but "practicality beats purity". If it's equally easy to pass in all dependencies explicitly to the configuration phase, then please do that.

Sandboxing

By setting `LD_PRELOAD` it is possible to override `libc.so` and filter all filesystem calls in order to create a sandbox and make sure that the build does not read from `/usr` (or anywhere outside the HashDist store, except through symlinks), which would indicate that the build reaches out to pull stuff it shouldn't. The Grand Unified Builder (GUB) takes this approach.

We may provide a `hit sandbox` command to do this. One may either want to turn that one for debugging, or all the time. One may have to create wrappers scripts around `gcc` etc. to set up sandboxing since some build tools like waf and scons like to control all the environment variables during the build.

Scheduler

To do many build artifacts in the right order (and in parallel in the right way), we should include a basic job scheduler for doing downloading and building. Say, `hit runtasks tasks.json`, with `tasks.json` thus:

```
{
    "numpysources" : {
        "type" : "fetch",
        "url" : "git://github.com/numpy/numpy.git"
        "hash" : "git:9c5a9226e7d742e3549d4e53d07d53517096f123"
    },
    "numpy" : {
        "type" : "build",
        "ncores" : 1,
        "body" : {
            "name" : "numpy",
```

```
            "dependencies" : {
                "blas" : "$atlas"
                ...
            },
            "sources" : {
                "numpy" : "git:9c5a9226e7d742e3549d4e53d07d53517096f123"
            }
        }
    }
    "atlas" : {
        "type" : "build",
        "exclusive" : true,
        ...
    }
```

Open question: Support the `"$atlas"` notation used above, or require that the hash for the atlas build section is computed and use that? Probably the latter?

Profile tools

A (software) "profile" is a directory structure ready for use through `$PATH`, containing subdirectories `bin`, `lib`, and so on which links *all* the software in a given software stack/profile.

Creating a profile is done by:

```
hit makeprofile /home/dagss/profiles/myprofile numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo ...
```

The command takes a list of artifacts, and reads `install.json` in each one and use the information to generate the profile. While the `install.json` file is generated during the build process, the Builder component has no direct knowledge of it, and we document it below.

Profiles are used as follows:

```
# Simply print environment changes needed for my current shell
$ hit env /home/dagss/profiles/myprofile
export PATH="/home/dagss/profiles/myprofile/bin:$PATH"

# Start new shell of the default type with profile
$ hit shell /home/dagss/profiles/myprofile

# Import settings to my current shell
$ source <(hit env /home/dagss/profiles/myprofile)
```

Of course, power users will put commands using these in their `~/.bashrc` or similar.

install.json

The `install.json` file is located at the root of the build artifact path, and should be generated (by packages meant to be used by the profile component) as part of the build.

Packages have two main strategies for installing themselves into a profile:

- **Strongly recommended:** Do an in-place install during the build, and let the installation phase consist of setting up symlinks in the profile
- Alternatively: Leave the build as a build-phase, and run the install at profile creation time

The reason for the strong recommendation is that as part of the build, a lot of temporary build profiles may be created (`hit makebuildprofile`). Also, there's the question of disk usage. However, distributions that are careful about constructing builds with full dependency injection may more easily go for the second option, in particular if the system is intended to create non-artifact profiles (see below).

The recommended use of `install.json` is:

```
{
    "runtime-dependencies" : {
        "python" : "python-2.7-io-lizHjC4h8z5e2Q00Ag9xUvus",
        "numpy" : "numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo"
    },
    "command" : ["hit", "install-artifact"],
    "profile-env-vars" : {
        "FOO_SOFT_TYPE" : "FROBNIFICATOR",
    },
    "parameters" : {
        "rules" : [
            ["symlink", "**"], # ant-style globs
            ["copy", "/bin/i-will-look-at-my-realpath-and-act-on-it"],
            # "/build.json", "/install.json" excluded by default
        ]
    }
}
```

(In fact, `python` is one such binary that benefits from being copied rather than symlinked.) However, one may also do the discouraged version:

```
{
    "runtime-dependencies" : {
        "python" : "python-2.7-io-lizHjC4h8z5e2Q00Ag9xUvus",
        "numpy" : "numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo"
    },
    "command" : ["$python/bin/python", "setup.py", "install", "--prefix=$profiletarget"],
    "parameters" : {
        "free-form" : ["json", "again", "; format is determined by command in question"]
    }
}
```

More points:

- The *runtime-dependencies* are used during the `hit makeprofile` process to recursively include dependencies in the profile.
- The *profile-env-vars* are exported in the `hit env`. This happens through a `profile.json` that is written to the profile directory by `hit makeprofile`. This can be used to, e.g., set up `PYTHONPATH` to point directly to artifacts rather than symlinking them into the final profile.
- `install.json` does not need to be hashed at any point.

Artifact profiles vs. non-artifact profiles

Usually, one will want to run `hashdist makeprofile` as part of a build, so that the profile itself is cached:

```
{
    "name" : "profile",
    "build-dependencies" : {
         "numpy" : "numpy-2.6-Ymm0C_HRoH0HxNM9snC3lvcIkMo",
     },
```

```
    "command" : ["hit", "makeprofile", "$numpy"],
}
```

Then, one force-symlinks to the resulting profile:

```
$ hit build < profile.json
profile-Z8GcCVzYOOH97n-ZC6qhfQhciCI
$ ln -sf $(hit resolve profile-Z8GcCVzYOOH97n-ZC6qhfQhciCI) /home/dagss/profiles/default
```

This allows atomic upgrades of a user's profile, and leaves the possibility of instant rollback to the old profile.

However, it is possible to just create a profile directly. This works especially well together with the `--no-artifact-symlinks` flag:

```
$ hit makeprofile --no-artifact-symlinks /path/to/profile artifact-ids...
```

Then one gets a more traditional fully editable profile, at the cost of some extra disk usage. One particular usage simply clones a profile that has been built as an artifact:

```
$ hit makeprofile --no-artifact-symlinks /path/to/profile profile-Z8GcCVzYOOH97n-ZC6qhfQhciCI
```

This works because `hit makeprofile` emits an `install.json` that repeats the process of creating itself (profile creation is idempotent, sort of).

The shared profile manager

To use a profile located in the current directory, `./myprofile` must be used. Calling `hit env myprofile` instead looks up a central list of profile nicknames. In `~/.hitconfig`:

```
[hashdist]
    profiles = ~/.hashdist/profiles # this is the default
```

...and following that, we find:

```
$ ls -la ~/.hashdist/profiles
myprofile -> /home/dagss/profiles/myprofile
qsnake -> /home/dagss/opt/qsnake
qsnake_previous -> ./artifacts/qsnakeprofile/0.2/io-lizHjC4h8z5e2Q00Ag9xUvus
```

(The paths in `/home/dagss` are likely further symlinks into `~/.hashdist/artifacts` too, but which artifact gets changed by the distribution). Distributions are encouraged to make use of this feature so that one can do:

```
$ hit shell sage
$ hit shell qsnake
```

...and so on. The intention is to slightly blur the line between different distributions; software distributions simply become mechanisms to build profiles.

2.4.2 Installing HashDist for use in software distributions

Dependencies

HashDist depends on Python 2.6+.

A bootstrap script should be made to facilitate installation everywhere...

Bundling vs. sharing

HashDist (by which we mean both the library/programs and the source and build artifact directories on disk) can either be shared between distributions or isolated; thus one may have build artifacts in `~/.hashdist` which are shared between PyHPC and QSnake, while Sage has its own HashDist store in `~/path/to/sage/store`.

Note: The main point of sharing HashDist is actually to share it between different versions of the same distribution; i.e., two different QSnake versions may be located in different paths on disk, but if they use the global HashDist they will share the build artifacts they have in common.

Another advantage is simply sharing the source store among distributions which build many of the same source tarballs and git repos.

Managing conflicting HashDist installations

By default, HashDist core tools are exposed through the `hit` command and the `hashdist` Python package. It is configured through `~/.hitconfig`:

```
[hashdist]
    hit = ~/.hashdist/bin/hit
    python2-path = ~/.hashdist/lib/python2

[sources]
    store = ~/.hashdist/source
    keep-transient = 1 week
    keep-targz = forever
    keep-git = forever

<...>
```

If a software distribution bundles its own isolated HashDist environment then the `hit` command should be rebranded (e.g., `qsnake-hit`), and it should read a different configuration file. Similarly, the Python package should be rebranded (e.g., `qsnake.hashdist`).

The command `hit` should *always* read `~/.hitconfig` (or the configuration file specified on the command line) and launch the command found there under the *hit* key. Similarly, `import hashdist` should add the package from the location specified in the configuration file to `sys.path` and then do `from hashdistlib import *`. The reason is simply that the paths mentioned in that file are managed by a particular version of hashdist, and we want an upgrade path. Essentially, the `hit` command-line tool and the `hashdist` Python package are not part of the software stack the distribution provides (unless rebranded). If you put an old, outdated profile in `$PATH`, the `hit` command found in it will simply read `~/.hitconfig` and then launch a newer version of `hit`. (However, `qsnake-hit` is free to behave however it likes.)

The best way of distributing HashDist is in fact to get it through the operating system package manager. In that case, the *hit* key in `~/.hitconfig` will point to `/usr/bin/hit`. Alternatively, a bootstrapping solution is provided and recommended which make sure that each distribution using a non-rebranded HashDist use the same one.

Bootstrap script

TODO

2.4.3 Profile specification layer

Nobody wants to use the core tools directly and copy and paste artifact IDs (unless they are debugging and developing packages). This layer is one example of an API that can be used to drive `hit fetch`, `hit build` and `hit makeprofile`. Skipping this layer is encouraged if it makes more sense for your application.

Included: The ability to programatically define a desired software profile/"stack", and automatically download and build the packages with minimum hassle. Patterns for using the lower-level `hit` command (e.g., standardize on symlink-based artifact profiles).

Excluded: Any use of package metadata or a central package repository to automatically resolve dependencies. (Some limited use of metadata to get software versions and so on may still be included.)

This layer can be used in two modes:

- As an API to help implementing the final UI
- Directly by power-users who don't mind manually specifying everything to great detail

The API will be demonstrated by an example from the latter usecase.

Package class

At the basic level, we provide utilites that knows how to build packages and inject dependencies. Under the hood this happens by generating the necesarry JSON files (including the build setup, which is the hard part) and calling `hit build` and `hit makeprofile`.

Note: This has some overlap with Buildout. We should investigate using the Buildout API for the various package builders.

Warning: A lot in the below block is overly simplified in terms of what's required for each package to build. Consider it a sketch.

Assume one creates the following "profile-script" where pretty much everything is done manually:

```
import hashdist
from hashdist import package as pkg

from_host = pkg.UseFromHostPackage(['gcc', 'python', 'bash'])

ATLAS = pkg.ConfigureMakeInstallPackage('http://downloads.sourceforge.net/project/math-atlas/Stable/3
                                        build_deps=dict(gcc=from_host, bash=from_host))
numpy = pkg.DistutilsPackage('git://github.com/numpy/numpy.git',
                             build_deps=dict(python=from_host, gcc=from_host, blas=ATLAS),
                             run_deps=dict(python=from_host, blas=ATLAS),
                             ATLAS=ATLAS.path('lib'),
                             CFLAGS=['-O0', '-g'])
profile = hashdist.Profile([numpy])
hashdist.command_line(profile)
```

Everything here is *lazy* (one instantiates *descriptors* of packages only); each package object is immutable and just stores information about what it is and describes the dependency DAG. E.g., `ATLAS.path('lib')` doesn't actually resolve any paths, it just returns a symbolic object which during the build will be able to resolve the path.

Running the script produces a command-line with several options, a typical run would be:

```
python theprofilescript.py update ~/mystack
```

This:

1. Walks the dependency DAG and for each component generates a `build.json` and calls `hit build`, often hitting the cache
2. Builds a profile and does `ln -sf` to atomically update `~/mystack` (which is a symlink).

Package repositories

Given the above it makes sense to then make APIs which are essentially package object factories, and which are aware of various package sources. Like before, everything should be lazy/descriptive. Sketch:

```
import hashdist

# the package environment has a list of sources to consider for packages;
# which will be searched in the order provided
env = hashdist.PackageEnvironment(sources=[
    hashdist.SystemPackageProvider('system'),
    hashdist.SpkgPackageProvider('qsnake', '/home/dagss/qsnake/spkgs'),
    hashdist.PyPIPackageProvider('pypi', 'http://pypi.python.org')
])

# The environment also stores default arguments. env is immutable, so we
# modify by making a copy
env = env.copy_with(CFLAGS=['-O0', '-g'])

# Set up some compilers; insist that they are found on the 'system' source
# (do not build them)
intel = env.pkgs.intel_compilers(from='system')
gcc = env.pkgs.gnu_compilers(from='system')

# env.pkgs.__getattr__ "instantiates" software. The result is simply a symbolic
# node in a build dependency graph; nothing is resolved until an actual build
# is invoked
blas = env.pkgs.reference_blas(compiler=intel)
# or: blas = env.pkgs.ATLAS(version='3.8.4', compiler=intel)
# or: blas = env.pkgs.ATLAS(version='3.8.4', from='system',
#                           libpath='/sysadmins/stupid/path/for/ATLAS')

python = env.pkgs.python()
petsc = env.pkgs.petsc(from='qsnake', blas=blas, compiler=intel)
petsc4py = env.pkgs.petsc4py(from='qsnake', petsc=petsc, compiler=gcc, python=python)
numpy = env.pkgs.numpy(python=python, blas=blas, compiler=intel, CFLAGS='-O2')
jinja2 = env.pkgs.jinja2(python=python)

profile = hashdist.profile([python, petsc, numpy, jinja2])
hashdist.command_line(profile)
```

2.4.4 User interface ideas

As mentioned on the Profile Specification Layer page, it lacks the ability to do automatic dependency resolution given package metadata. That is some of what belongs on this level.

On the other hand, for some usecases the typical automatic solution is too much, and a more explicit and less traditional model may be better.

The explicit dict-of-git-repositories model

TODO

The package-repository and metadata-driven model

TODO

h

h

L

M

P

R

S

T

U