There's a lot of interest in improving the performance of the Meteor command-line tool and build system, and if you're interesting in helping, this document is meant to be a jumping off point.

Sooner or later, the tool will be inevitably be substantially rewritten or replaced in order to achieve those sweet sub-second build times everyone loves, because it is currently too complex and reliant on the file system to get there. However, until then, there's work that can be done to make things faster.

For more on the tool, see also the README files in this directory and its sub-directories.

Profiling

The best way to get information about where time is going in the Meteor tool is with built-in profiler, activated with the environment variable `METEOR_PROFILE`. The value is interpreted as a threshold in ms, so the value `1` means to report on all calls that take more than 1ms:

```
METEOR_PROFILE=1 meteor ...
```

You'll then see a report (or a series of reports) printed to the console that might look something like:

```
| (#1) Profiling: ProjectContext prepareProjectForBuild
| ProjectContext prepareProjectForBuild.........9,207 ms (1)
|    _initializeCatalog.............................24 ms (1)
|       files.readFile                               7 ms (2)
|       runJavaScript package.js                     2 ms (1)
|       files.rm_recursive                           4 ms (4)
|       other _initializeCatalog                    11 ms
|    _resolveConstraints........................6,702 ms (1)
|       bundler.readJsImage.........................42 ms (1)
...
| (#1) Total: 9,544 ms (ProjectContext prepareProjectForBuild)
```

The report is a simple top-down listing, where each entry represents a function call or other instrumented block in the code. A dotted line is drawn for entries that have child entries, and the total time of the child entries should add up to the parent entry (though turning up the profiling threshold may mess with the math). The number in parentheses is the number of times that function or block was called, and the time listed is the total cumulative time of all the calls.

Entries in the profiler come from annotations that instrument code, and unlike what you get with sampling profilers (like `node --prof` or dtrace), the call counts are exact. All times are measured with high-resolution wall-clock timers and only rounded for display. However, time measurements are quite susceptible to being inflated by random acts of garbage collection; profiler overhead; competition for the CPU; and so on. Also, if the tool were more highly parallelized, these

stopwatch-type measurements might stop adding up, but so far they've been pretty accurate as far as we know.

Please see the section "Performance Considerations" before interpreting a profiler report.

Performance Considerations

Read this section before drawing conclusions or comparing numbers online.

The Meteor build tool is complex and has several layers of caches and modes of operation. It's not apples-to-apples to compare a rebuild time between a Meteor release and a checkout, for example, or to compare a start-up with a cold cache to one with a warm cache. This section lists the various nuances of build tool performance.

Unreported Work

Not everything is covered by the reports we generate ("prepare project" and "build app"). For example, you won't see:

- Time spent in app start-up
- Time spent creating file watchers after a rebuild

Caching

The tool saves work using various disk caches and in-memory caches. There are caches of:

- Built packages and plugins
- Compiler plugin results - and not just on the most recent version of a file! If you are testing rebuilds by toggling a character back and forth, you may hit cache.
- Linker output
- The entire server program, for client-only changes
- Constraint solver results

Therefore, app start-up will be slow if the disk caches haven't been generated or are invalid because a package needs to be rebuilt, for instance, or the constraint solver needs to run. Rebuilds are much faster than the initial build of the app, and the second rebuild even seems to be faster than the first, at which point the time levels off.

See the section "More About Caches".

Release vs. Checked-Out Branch

The Meteor tool behaves differently depending on whether you are running a Meteor release, or a checked-out branch of a git repository. Most people run a Meteor release, but if you are making any changes to the tool or core

packages, you'll be running Meteor in checked-out mode for testing, debugging, and measurement, so it's important to know the discrepancies. (See also "More About Release vs. Branch".)

The following are performance-related differences between running an app with a release of Meteor vs. a checked-out branch of Meteor:

- Checked-out Meteor recompiles core packages. When you run an app using checked-out Meteor, it will compile core packages into `APP/.meteor/local/isopacks` – that's right, into your app! This process can take a while – as much as a couple minutes – but it's only done once per package per app, unless the package changes. When the app starts up, it will read in the built packages and decide if it needs to do anything. However, there's noticeable extra time "preparing" for each rebuild, because the core packages are rescanned, and the large number of files being watched for changes can cause several seconds of delay after a rebuild as well.
- Constraint-solver performance is different. Core packages are constrained by a release and also by running from checkout, but the effect is different, and the constraint solver seems to have an easier time on the constraint problems it gets when running from checkout, because of the way core packages are pinned like local packages.
- Constraint-solver won't save `APP/.meteor/versions` if there's a release in `APP/.meteor/release`. Normally, when running released Meteor, `APP/.meteor/versions` provide stability to your package versions between runs of your app. However, when you run `REPO/meteor`, core package versions are temporarily overridden by the versions in the repo, but not saved, unless `APP/.meteor/release` is set to `none`, meaning the app is not written against a released version of Meteor. This difference comes up when testing and measuring the behavior of the constraint solver.

Time spent rebundling and rebuilding the app seems pretty similar across Release and Checkout.

App Differences

Different apps can strain different parts of the Meteor tool, such as:

- Constraint solver - lots of dependencies; large package graph
- File watchers - lots of total files, e.g. thousands
- Modules - as of Meteor 1.3, imports from packages and `node_modules` must be scanned and analyzed
- CSS processing - e.g. in case of a large LESS or Sass framework in the app
- Community plugins, including legacy plugins without caching between builds

Hardware and OS

Spinning disks are much slower than SSDs.

Windows has a very different file system from Mac OS X or Linux.

On OS X and Linux, we have all these nice properties: * Usable symlinks * Renaming a file is cheap * Overwriting a file by rename is atomic * An file can be deleted ("unlinked") while open * A file can be moved (renamed) while open

Windows doesn't have these properties; for example, renaming a file is a copy operation. In addition, virus scanners and other software will sometimes block or postpone file operations. And, of course, there are the path separator differences. Parts of the tool work really hard to target OS X and Linux efficiently (with symlinks and renames, for example) while doing something that behaves well on Windows, even if it's slower.

Instrumenting Code for the Profiler

If you see a large time next to "other (function)", that's a case where the tool needs further instrumentation. For example, you may see something like this:

```
| ImportScanner#_readFile..............1,338 ms (329)
|    files.readFile                       100 ms (329)
|    sha1                                   3 ms (329)
|    other ImportScanner#_readFile      1,235 ms
```

In this example, some function calls under `ImportScanner#_readFile` just happen to be instrumented (`files.readFile` and `sha1`), but no other calls under `ImportScanner#_readFile` have been annotated yet. Given how much time is spent in this call, we should seek out more calls to annotate. You can determine by experimentation which calls are significant enough to add to the profile.

To annotate a function or block in tool code to be included in the profile, use one of the following forms:

```
// Profile(.) takes a function and wraps it, preserving arguments
// and this.  You can often get away without introducing any extra
// indentation in your text editor.
var doIt = Profile("doIt", function (...) {
  ...
});

// Wrap a method
MyClass.prototype.doIt =
  Profile("MyClass doIt", MyClass.prototype.doIt);

// Profile.time(.) invokes a block immediately and times it,
// passing the return value through.
```

```
function doIt() {
  ...
  var result = Profile.time("doIt", () => {
    ...
  });
  ...
}
```

The Profiler is activated using `Profile.run(.)`, which will cause a report to be generated. Currently we generate a couple reports when `METEOR_PROFILE` is set – one for preparing the project (including constraint solving) and one for building or rebuilding the app.

In addition to tool code, you can also use `Profile` in compiler plugins. If you want to use `Profile` in packages that are loaded into the tool (e.g. packages depended on by compiler plugins, or specially loaded into the tool as isopackets), you should always test `(typeof Profile !== 'undefined')` before accessing `Profile`, or pass it in from the tool as an option.

Known Areas for Improvement

These are areas to improve or investigate, along with what's known, in no particular order.

CSS Minification

See: https://github.com/meteor/meteor/pull/5875.

Meteor's CSS minification is user-pluggable since 1.2. What the standard minifier does is concatenate your CSS into one file with `import` statements hoisted to the top. In production mode, this file is minified before being served. The benefit of doing the same CSS merging in development and production is that the meanings of relative paths in CSS files and the order of rules doesn't change. However, reparsing and merging the CSS for a typical app can take anywhere from 500ms to a couple seconds or more, depending on the amount of CSS.

Initial profiling shows the time is split between different operations like parsing CSS, generating CSS, and working with source maps. Approaches could include:

- Inserting a new cache, such as around parsing files
- Only recalculating the CSS if it changes
- Using a faster CSS library

Source Maps

The tool spends noticeable building source maps, as do other tools such as Webpack, which have evolved creative mechanisms to cut down the time. There's some question of whether the `source-map` library is as efficient as it could be, and whether we're using it as efficiently as we could be.

Linker Cache

See: https://github.com/meteor/meteor/issues/5818

The "linker cache" on disk at `APP/.meteor/local/bundler-cache/linker` has some very large files that take a long time write, and they are apparently not cleaned up either. Sometimes you will see calls to `files.writeFileAtomically` taking many seconds at the top of the "Rebuild App" report, but somehow outside the top level (probably because they are fired off asynchronously).

Constraint Solving

Some initial work has been done to make constraint solving fast in the typical case where a previous solution exists in `APP/.meteor/versions`, but there is possible room for improvement.

For example, the previous solution could be checked without invoking the logic solver at all, though the manual logic would have to match closely. Implemented!

Time spent reading from the packages database (a SQLite file) could be understood better and improved by batching queries, reading less data, or using SQLite better.

See: https://github.com/meteor/meteor/issues/6137

"Updating package catalog. . . "

Talking to the package server to download new package metadata seems to take longer than it should; easily 5 seconds, based on watching the progress display, and occasionally much longer.

Hoisting Build Work

You'll find scattered through the tool code comments about how it would be more efficient to do certain processing or reading in a way such that it wouldn't have to be done a second time in a different part of the tool, or it wouldn't have to be done on each rebuild. The first one is about the flow of information in the tool, and the second is about making better use of information about what changed, either through caching or file-watching.

File watching

It seems like it would be great if file-watching gave the tool perfect information, so it wouldn't have to go to disk and read lots of files even though, in theory, if they had changed it would know. You can imagine a tool that is super fast because it basically reads and writes diffs of the state of the file system. It's not clear how feasible that is, though. Right now, when we're told a file has changed, we immediately go to work, which includes tearing down file-watchers;

we don't wait to get the exact list of what files have changed. For the tool to remain receptive to changes would require a different paradigm.

However, it seems like some middle ground would be possible, where the tool does not simply merge all the lists of files it has ("WatchSets") and wait for any change to any of them, but where rather it responds to changes to different sets of files by rebuilding different parts and then putting them together; for example, only processing the CSS if some CSS changed.

It also might be that changing to a different file-watching library would have performance benefits.

More About Caches

Disk caches exist in the following locations, where `APP` is the app:

- `APP/.meteor/local/isopacks`
- `APP/.meteor/local/bundler-cache`
- `APP/.meteor/local/plugin-cache`
- `~/.meteor` (package storage, when running a release)
- `REPO/.meteor` (package storage, when running a checked-out branch)

You can safely remove these directories to clear your caches, but don't remove `APP/.meteor/local` if you have data in your app's database, because the local Mongo DB resides in `APP/.meteor/local/db`. (If you don't care about the DB, you can remove all of `local` for the ultimate clean checkout experience.) The package storage directory holds master copies of packages that will be copied or linked into your project (like Maven's `.m2`). There's probably no reason to clear them out, except to save some disk space. It's sometimes nice to know where these files are located in case you want to poke around or add `console.log`s to released code for debugging.

More About Release vs. Branch

Most people run a released version of the `meteor` tool, but if you are making any changes to the tool or core packages, you'll be running it in checked-out mode for testing and debugging. Here's how the two modes work:

- Release: This is when you run the installed `meteor` tool on your computer. The tool and core packages are part of a release with a name like `METEOR@1.2.1` and were downloaded from the package server and cached in `~/.meteor` along with other Atmosphere packages. When the tool is run, the current release is determined from `APP/.meteor/release` when running an app, or the value of `--release` on the command-line.
- Checked-Out Branch: This is when you check out the `meteor/meteor` GitHub repository as `REPO` and run `REPO/meteor`. The tool is loaded from `REPO/tools`, and core packages are loaded from `REPO/packages`. The tool takes care of transpiling itself and rebuilding out-of-date core packages.

Core packages in the repo behave much like local packages in your app; package `foo` is treated as the only known version of `foo`, and its files are watched for hot-reload.