Inside A Minecraft Launcher

Recently, I've been playing around with writing a Minecraft launcher from scratch, and also contributing to PolyMC and ManyMC. Through these harrowing interesting experiences, I have gathered a lot of knowledge about how Minecraft launchers actually work from the inside.

And I thought it would be nice to share this information with those of you who want to work on this topic yourself, since the current documentation isn't very complete.

Vanilla

Installing a vanilla version is the basis for everything else. I will write about installing loaders like Fabric and Forge later on, but this is the most basic thing launchers should do.

0. The working directory

The working directory depends on the system. In any case it is a specific directory reserved for Minecraft. For the official launcher this is one single directory, while in multi-instance launchers like MultiMC or HMCL, this directory can be customized according to the launcher. However, most of the files you download / generate will be in this directory.

This directory will henceforth be referred to as .minecraft for convenience. This is the directory used on Linux.

1. The Version Manifest

The version manifest is the single source of truth for what versions there are. The URL is https://launchermeta.mojang.com/mc/game/version_manifest.json, and the format is approximately like this:

Obviously, the latest key corresponds to the latest versions for releases and snapshots. In the versions list, there is an ID (the name of the version), the type (release or snapshot), the URL to the version data, and the time and release time (which don't really matter that much). The version data URL is going to be used in the next step.

2. The Version Data

After selecting the version, the JSON data from "url" would be downloaded to .minecraft/versions//.json. This version data JSON has a few sections of data:

"arguments" - arguments to pass

This contains the arguments to run the main JAR with. The "jvm" flags are to be provided before the main JAR, while the "game" flags are provided after the main JAR. So a launch

script would look something like this:

java -jar [jvm flags] [main class] [game flags]
The flags provided can either be a string or an object with the type

interface Rule
action: 'allow'

interface OptionalArgument string[];

The rules feature is also used in the downloading of libraries in a later step. For platform-specific arguments, there are a list of rules to match, for either operating system, architecture, or the feature (e.g. demo user).

Do note that there are a few substitutions needed in these arguments. For the JVM arguments, these are the substitutions needed:

$natives_directory: the directory of the platform natives (to be mentioned later on)

$launcher_name: the name of the launcher

$launcher_version: the version of the launcher

$classpath: a list of the paths of all the library JARs and the main JAR downloaded joined with :s

$classpath_separator: :

$primary_jar: the path to main JAR

$library_directory: the overall folder in which the libraries were downloaded

$game_directory: the “working directory” mentioned before

For game arguments, there are also some substitutions needed:

$auth_player_name: the username of the player

$version_name: the name of the version (e.g. 1.18.2)

$game_directory: same as JVM

$assets_root: the root folder of the downloaded assets, typically .minecraft/assets

$assets_index_name: the asset index version to use, typically the minor version (e.g. 1.18)

$auth_uuid: the authentication UUID provided by MSA

$user_type : all mojang nowadays

$version_type: the version type, release or snapshot (although this can display any string)

For the JVM args, you can also add any optimization arguments you like, such as enabling G1GC.

"assetIndex" - the index of assets to download

The textures and music and UI controls are all assets, and there is typically an asset index for each minor version. This object has the following type definition:

interface AssetIndex
id: string;
sha1: string;
size: number;
totalSize: number;
url: string;

where "sha1" is the SHA-1 hash of the file, "size" is the size of the JSON file, "totalSize" is the size of all the assets combined, and "url" is yet another JSON file to get for the assets.

"downloads" - the main JARs to download

This includes the JARs that you will actually call with java on launch. It’s an object containing downloads for both clients and servers in the format sha1: string; size: number; url: string; . Again, the SHA-1 hash is provided. The file in "url" is downloaded to .minecraft/versions//.jar. For clients, the file to download is ["downloads"]["client"]["url"].

"libraries" - the libraries to download

This is arguably the most complex part of the downloading process. It will be detailed in its own dedicated section in step 3.

"logging" - the logging configuration file to provide to log4j

logging:
client:
argument: string;

file:
id: string;
sha1: string;
size: number;
url: string;

type: 'log4j2-xml';

This contains an XML config file to provide to log4j. Download from ["logging"]["file"]["url"]. The argument in ["logging"]["client"]["argument"] should have the special keyword $path replaced with the path the config file was downloaded to. This can be anywhere, but typically in .minecraft/versions//log4j2.xml.

"mainClass": the main class to call in the execution of java

The others are not very important to our discussion here, and are pretty straightforward, so just look at the JSON if need be.

3. The Assets

As you may recall, in the "assetIndex" section there was a JSON file to download to .minecraft/assets/indexes/.json. This JSON file contains a single "objects" key in which there are asset files in the following scheme:

The name of the file does not matter. The file URL is derived from the hash as https://resources.download.minecraft.net//. This file is downloaded to .minecraft/assets/objects//, basically mirroring the URL structure.

4. The Libraries

This section downloads the libraries needed for the game. Each library is in this standardized format:

The LibraryDownload type is pretty similar to the ones before. The "rules" object functions identically to the one for JVM and game arguments. For most libraries, there will be an "artifact" key, and all you need is to download library.downloads.artifact.url to .minecraft/libraries/. However, for libraries with natives, it is a bit different.

Natives are binaries such as dynamic libraries and DLLs that are specific for each platform. If the "natives" key is present, the classifier needed to download will be in library["natives"][os]. Then put this classifier into library.downloads.classifiers[classifier], and download the JAR in there to a temporary location. Then, unzip (since JARs are zips) the JAR and take the natives (glob: *.dylib,so,dll) and put them in a directory for the version, typically .minecraft/versions//natives.

This directory is what you substitute $natives_directory for. This is necessary for some libraries like LWJGL and java-objc-bridge to work, since they have platform-specific bindings.

5. Actually launching

When you launch, simply run the substitutions on the JVM and game arguments previously mentioned. Then execute java like this:

And there you have it! Downloading and launching vanilla Minecraft.

Fabric / Quilt

For some players, modding makes the experience a lot better. One of these modding toolchains is Fabric, and it is also pretty easy to install on the basis of vanilla.

Quilt is a very recent fork of Fabric that aims to be more community-driven and contain more standardized features. The installation process is similar.

Fabric has an official website called meta.fabricmc.net that provides information on Fabric. For a specific vanilla version, you can open up https://meta.fabricmc.net/v2/versions/loader/ and it will give you a list of compatible Fabric Loader and Intermediary versions.

For Quilt, this would be https://meta.quiltmc.org/v3/versions/loader/.

In each object in the list, there is "loader", "intermediary", and "launcherMeta".

In launcherMeta.libraries there are a list of libraries to be appended to the vanilla library list. launcherMeta.libraries.common and launcherMeta.libraries.client are the ones needed for a client. The loader and intermediary also need to be appended.

In launcherMeta.mainClass, there are the custom main classes for Fabric or Quilt. Here, don’t use the officially provided main class to launch - use the one provided in launcherMeta.mainClass.client.

One thing to notice is that the library format provided by Fabric is different from the traditional Mojang one - it includes a "name" key and a "url" key. This is a reference to a Maven repository, the packaging format of Java. GAMES is in the format org1.org2:package:version.

You can easily derive the download URL from these two attributes by the following template:

After downloading the additional libraries and editing the mainClass, just launch the game with the same template as vanilla and a Fabric / Quilt modded client will be up and running!

Forge & Liteloader

I am still investigating how Forge and Liteloader work. Will post more information about it in a follow-up post.

MultiMC-style Launchers

MultiMC and forks of it use a custom format of their own JSON files called meta files. These meta files are generated from the sources cited above, such as launchermeta.mojang.com and meta.fabricmc.net. (Note how they all use the diction “meta”.) The MultiMC-style meta file is similar, but different from the vanilla ones.

Here is a table of the attributes from the MultiMC wiki:

The libraries format in MultiMC are much more standardized than the Mojang ones. Reference this for more details.

MultiMC has a concept of patches, in which these different attributes can inherit and override one another. “Minecraft” is a patch, derived from the launcher meta. (Also, LWJGL is removed from the Minecraft patch and placed in its separate patch, for some reason.) Following this pattern, Fabric Loader and Forge are also patches.

Each patch has a "uid" to uniquely identify themselves and be able to specify one another.

Patches can depend on one another - for instance, the Minecraft patch depends on the LWJGL patch for that specific version.

An example for Fabric Loader 0.14.3:

This standard makes downloading and launcher in the launcher a much smoother experience, and separates the concerns of parsing the data from various sources and actually doing the work.

As to the working directories, MultiMC uses shared folders for libraries and assets across instances, since libraries referred to with the same Maven identifier would probably be the same, and assets can be reused. The working directory is set to the individual instance folder, and because you can pass your own classpath and assets_root, it works flawlessly.

MultiMC uses its own custom NewLaunch.jar to launch, though. I’m not sure why that would be used.