Binned Incremental Font Transfer (IFTB)

July 25, 2023

Agenda

1. Technology review
    a. Static Subsetting
    b. Range Request IFT
    c. A few basics of subsetting
2. Binned Incremental Font Transfer (IFTB)
    a. File Formats
    b. Client-side operation
    c. Technology comparison
    d. IFTB encoding
    e. Preloading
3. Some miscellany
4. Resources
5. Questions and Discussion

Static Subsetting

a. A “source” font file is split into many subset files.
b. Each can render a particular set of unicode codepoints.
c. The `unicode-range` CSS property maps from codepoints to subset files.
d. The download efficiency depends on ability to group codepoints that tend to be needed together into the same file, so that many files are *not* needed.

1. Both glyph outline data and shaping data are subset
2. Processing is done in advance, allowing high compression levels (no real-time compression requirement) and avoiding dedicated server-side code.
3. Behavior of original file is not preserved (because shaping data that applies *between* subsets is thrown away).

Range Request IFT

a. Download portions of a font file rather than the whole file. In practice:
    i. Download all shaping data
    ii. Use shaping data and `cmap` (along with content to be rendered) to determine which glyphs are needed.
    iii. Download the data for those glyphs using offsets in `loca`, `gvar` or the `CFF` CharStrings index.

b. More efficient when glyphs that tend to be needed together are adjacent in the file.

1. Only glyph path data is subset.
2. Server-side data starts uncompressed, requested ranges can be dynamically compressed by server before downloading.
3. Behavior of original font file is preserved.
4. Too many network round-trips for composite/component glyphs.

Concerns with Range Request

1. Tricky client-side implementation. Steps are:
   a. Download shaping data
   b. Run shaping steps to determine glyphs needed.
   c. Pause to download data for those glyphs.
   d. Continue glyph rendering process.
2. Appears to require at least three round trips for initial load (perhaps two can be concurrent?):
   a. Initial request for shaping data
   b. CORS preflight for multi-range request
   c. Multi-range request for glyph data.
   d. Extra round-trip for variable TTF (`gvar` table header)
3. Choice between guessing initial length and resetting connection.

Some basics of subsetting

- Fonts are typically subset in two stages:
    1. Taking a set of codepoints and layout features as input, determine the set of glyphs needed to render any combination of those parameters.
    2. Rewrite the various OpenType tables of the font to only refer to glyphs in that set.
- The first stage suggests a sort of mapping between input parameter set and glyph set.
- I will call the latter the “glyph closure” of the parameter set.

Binned Incremental Font Transfer (IFTB) Font Formats

- Two formats
  1. Variant of OTF, with `_iftb.otf` or `_iftb.ttf` file suffix.
  2. Variant of WOFF2, with `_iftb.woff2` file suffix.
- Each is the same as its counterpart except:
  a. Glyph path data is “sparse” (empty or minimal entries for most glyphs).
  b. Extra `IFTB` font table.
  c. OTF version is `IFTB` rather than `OTTO` or `0x00010000`.
  d. Some additional table and subtable ordering requirements.
- Can (pretty much) switch between formats with standard WOFF2 compress/decompress

IFTB “bins”

- Each IFTB-encoded font has a set of bins, indexed from 0.
- Each GID in the font is mapped to at least one bin.
- There are also per-layout-feature mappings from bin to bin. These are referenced when the indicated layout feature is enabled.
- Both types of mapping are stored in subtables of the `IFTB` table.
- Bins are analogous to the set of glyphs in a static subset.
- The outline data for all glyphs assigned to bin 0 are always included in the initially downloaded font file.

IFTB "chunks"

- Each bin (other than 0) has an associated chunk file.
- There are uncompressed and compressed chunk files, that start with bytes `IFTC` and `IFTZ` respectively.
- Each chunk file contains the glyph outline data for every GID assigned to is bin.
- Header information indicates what table(s) each GID is taken from.
  - Usually one table: `glyf`, `CFF` or `CFF2`
  - Sometimes two tables: `glyf` and `gvar`.

Client-side operation

1. Download initial IFTB-encoded font file (and notices its OTF version). If WOFF2, decompress.
2. Map needed codepoints to GIDS using the font's `cmap` table (Format 4 or 12)
3. Compute, in several stages, the set of bins needed:
   a. Map each GID from last step to bin using IFTB GID map, add bin to set.
   b. Sort list of needed layout features and for each, check if there is a subtable for that feature. If there is, use the bin-to-bin mapping with the current bin set to add bins for that feature.
   c. Subtract any bins from the set that are already merged (using chunk bitmap in IFTB table).
4. Download the chunk file for each missing bin.
   a. Relative chunk URI template string in IFTB table.
   b. Make specific to chunk by adding hexadecimal characters of glyph index.
   c. Make absolute by reference to URL of original font file.
5. As chunks arrive, decompress and extract glyph data into intermediate datastructure. When last chunk arrives, integrate all data into respective font tables.
6. Update chunk bitmap in IFTB table.
7. Set OTF version to `OTTO` or `0x0001000` and update sfnt-header checksums

Technology comparison

- Compression *level* of IFTB high like static subsetting, and higher than patch-subset or range request, because files are encoded in advance. (However, higher-level is partially offset by chunks being compressed independently.)
- IFTB *granularity* similar to static subsetting and coarser-grained than content-specific systems.
- All shaping data is always downloaded, as with range request
- Top-level client-side *interface* most like patch-subset.

CDN cache compatibility

- IFTB is very compatible with CDNs and caching, and needs no HTTP facilities beyond efficient multi-file downloads (i.e. HTTP 2 or higher).
  - “very compatible”: Chance of the initial and chunk files for a widely used font being cached is pretty good, because just those files serve all requests.
- Choice of technology relative to network bottleneck:
  - Low bandwidth or congested “last mile”: patch-subset will generally work better.
  - “last mile” is OK, original server is “distant”, and font is widely used: IFTB will often work better.
  - When chance of caching either is low: patch-subset will generally work better.

Shaping and rendering behavior

IFT technology *should* provide the same shaping and rendering behavior as the original font.

- Patch-subset does so by carefully constructing one self-consistent font subset.
- Range request does so because it always downloads all shaping data and then determines which glyphs are needed.
- Static subsetting does not provide the same behavior.

The IFTB encoder and rendering behavior

IFTB also has all shaping data. However, the set of glyphs available is determined by the set of bins that have been merged.

Ensuring the same behavior therefore falls on the IFTB encoder, which must meet this “closure requirement”:

- The set of glyphs contained in a set of bins computed from the input parameters — the codepoint and layout feature sets — must be a superset of the glyph closure for those parameters.

Encoding “options”

Simple cases of encoding:

a. A glyph that *only* has a code point can be added to whatever bin makes sense for codepoint grouping.
b. A glyph only substituted under one circumstance can be assigned to the bin containing a glyph it substitutes for.

Options for more complicated cases, where a glyph is needed in more than one “proto-bin”:

1. Merge two proto-bins together.
    - Good: No data duplication, doesn’t increase size of base file.
    - Bad: Can wind up with too many unrelated codepoints in a bin
2. Duplicate glyph across bins:
    - Good: Doesn’t result in grouping unrelated codepoints
    - Bad: Potentially increases total download size
3. Put glyph in bin 0:
    - Good: No duplication, easy “out” for complicated cases.
    - Bad: Glyph always downloaded

The prototype encoder

The current prototype encoder:

- Is quite primitive
- Is designed to, and appears to, meet the closure requirement for any relevant font. (Only `glyf`/`gvar` or `CFF`/`CFF2`.)
- Is based off HarfBuzz subsetting logic.
- *Never* duplicates a glyph.
- Puts anything “complicated” into bin 0. (“Is substituted for another glyph and also has its own codepoint” can count as “complicated”.)
- Biggest problem in practice/lowest hanging fruit: `aalt` and `nalt` layout features.
- Even so, it works surprisingly well on many CJK fonts.

A better encoder

- I believe an encoder that will work quite well across the needed use-cases (e.g. almost all existing CJK fonts) is possible, just based on the available encoding options.
- I also believe it is a tractable project.
- There *may* be a chicken-and-egg problem arguing for IFTB before building a good encoder.
  - Not needed to write the specification, but helpful for performance data.
- Unfortunately, making a better encoder, or improving IFTB more generally, is not my current project at work.

Preloading

- There is overhead in downloading chunk files. (HTTP headers, chunk metadata.)
- As noted, the IFTB table contains a “chunk bitmap” indicating what chunks are already present.
- The bitmap is normally updated on the client side, but can also be updated on the server side, by “preloading”:
  - Integrate some set of chunks into a copy of the initial file (beyond bin 0).
  - WOFF2-compress that file and give it its own name.
  - Example: Initial IFTB font file preloaded with high-frequency Japanese glyphs.
- Preloading decreases the number of bits downloaded:
  - No duplication among glyphs in preloaded bins.
  - All preloaded data is Brotli-compressed as a unit.
- Different preloaded files still share the same set of chunks for downloading further glyphs.
  - Fewer files to store/register with CDN.
  - Sharing chunks between scripts means higher chance of a given chunk being cached.
- Allows some documents to render with just one round-trip.

Miscellany

- (Probably) abandoned range-request-based chunk retrieval option
- Size and extent of GID to chunk map in IFTB table
- Modest HarfBuzz library modifications for prototype:
    - https://github.com/skef/harfbuzz/tree/chunkmods
- Trivial WOFF2 encoder library modifications for prototype:
    - https://github.com/skef/woff2/tree/iftb_changes

Resources

- Prototype repository: https://github.com/adobe/binned-ift-reference
  - Released under W3C Software License
  - Includes very simple demo
- Proto-specification branch: https://github.com/skef/IFT/tree/iftb
  - Based on March 2023 commit (before division into two the protocols).
  - Really more a set of notes than a proper attempt at a PR
  - More or less accurate (as I remember)

Questions/Discussion

- Break?
- Questions/discussion