日本語文字コード入門α版

歴史とUnicodeにおける実装

楠 正憲

2025年5月23日

目次

はじめに

本書は「公共情報システム開発のための日本語文字コード入門」と題し、公共情報システムの開発に携わるエンジニア、地方公共団体職員、そして日本語文字コードに関心を持つすべての方々に向けて書かれた入門書である。

> **注意:** 本文書はLLMツール等を用いて作成された文章をベースに、内容の正確性について順次確認と修正を進めているものです。誤りや不正確な記述が含まれている可能性があるため、現時点での記述を過度に信頼せず、必要に応じて一次資料を参照いただくようお願いいたします。誤りを見つけられた場合は、奥付の連絡先までご一報ください。

コンピュータで日本語を扱う上で、文字コードは避けて通れない存在である。特に、公共情報システムにおいては、氏名、住所など、正確な文字情報が極めて重要であり、文字コードに関する正しい理解は、システムの品質を左右する重要な要素となる。

本書では、日本語文字コードに焦点を当て、その歴史的変遷から、技術的な仕組み、さらにはシステム開発における実践的な知識までを、体系的に解説する。

本書を読む前に

文字コードは、一見すると地味で目立たない技術かもしれない。しかし、コンピュータが文字を扱うための基盤であり、その仕組みを理解することは、システム開発において非常に重要である。特に、日本語は、漢字、ひらがな、カタカナといった多様な文字種、膨大な数の漢字、そして異体字の存在など、他言語に比べて複雑な文字体系を持っている。そのため、日本語を扱うシステム開発では、文字コードへの深い理解が不可欠である。

本書は、文字コードに関する事前知識がない方でも理解できるよう、基本的な用語から丁寧に解説している。また、既に文字コードに関する知識を持つ方にとっても、日本の文字コードの全体像を体系的に理解し、実務上の課題を解決するためのヒントを得られる内容となっている。

対象読者と本書の目的

本書は、以下のような読者を対象としている。

- 公共情報システムの開発・運用・保守に携わるエンジニア
- 地方公共団体で自治体システム標準化や情報システムを担当する職員
- ベンダー企業で公共情報システム関連の製品・サービスに携わる方
- 公共分野における多言語対応に関心のある方
- 日本語文字コードに関心を持ち、知識を深めたいと考えている方

本書の目的は、読者が日本語文字コードに関する正しい知識と理解を持ち、公共情報システムの開発・運用をより円滑かつ正確に遂行できるよう支援することである。また、文字コードの問題を通じて、情報システムにおけるデータ品質の重要性について改めて考えるきっかけを提供したいと考えている。

本書の特徴と構成

本書では、まず文字コードの基本的な概念を解説し、次に日本語文字コードの歴史的変遷を概観する。そして、主要な文字セットの詳細、Unicodeの仕組みと多言語対応、さらには各種ソフトウェアにおける文字コードの実装方法、そして公共情報システム開発における実践的な課題と対応策について解説する。

特に、以下の点に重点を置いて解説を行う。

- **歴史的変遷**: 日本語文字コードがどのように発展してきたのか、その歴史的背景を理解することで、現在の状況をより深く理解することができる。
- **技術的仕組み**: 文字集合、符号化方式、サロゲートペア、IVS、結合文字など、文字コードを理解する上で重要な技術的仕組みを、図表を交えて分かりやすく解説する。
- **行政における標準化の取り組み**: 戸籍統一文字、住基ネット統一文字、文字情報基盤、そして行政事務標準文字など、行政における文字コード標準化の取り組みについて、その背景や意義、現状を詳述する。
- **公共情報システム開発における実践的知識**: Unicode対応、外字処理、データベースとの連携など、公共情報システム開発において直面する課題と、その対応策について具体的な例を挙げて解説する。

公共情報システムと文字コード

公共情報システムは、住民の生活に密接に関わる重要な社会インフラである。特に、戸籍、住民基本台帳、税務、社会保障など、人々の権利や義務に関わるシステムでは、氏名、住所などの文字情報が正確に扱われることが不可欠である。

しかし、前述の通り、日本語の文字体系は複雑であり、その電子化には多くの困難が伴った。そのような中、関係者の長年の努力により、文字コードの標準化が進められてきた。JIS規格の策定、Unicodeの登場、そして行政機関における様々な取り組みを通じて、日本語をコンピュータで扱うための基盤が整備されてきたのである。

本書では、このような文字コード標準化の歴史的意義を振り返りながら、現代の公共情報システム開発における文字コードの扱いについて詳述する。そして、本書が、公共情報システムの品質改善と、住民サービスの向上に貢献することを願っている。

文字コードの基礎知識

本章では、計算機で文字を扱う際に必要となる基礎知識について説明する。まず、文字、符号、コードなどの重要な用語を定義し、文字コードの基本的な仕組みを解説する。次に、文字を表現するフォントデータの形式や、字形、字体といった、文字のデザインや見た目に関する概念についても説明する。さらに異なる文字集合間での変換時に生じる縮退という現象、および複数の文字を組み合わせて表示する組み文字という技法についても言及する。これらの基礎知識は、続く章で文字コードの歴史や具体的な仕組み、Unicodeを理解する上で重要な土台となる。

文字、符号、コード：用語の定義

本節では、本書で用いる重要な用語である「文字」「符号」「コード」について、それぞれの定義を明確にし、相互の違いを説明する。これらの用語は、日常会話ではしばしば混同されがちだが、情報処理、特に文字コードを扱う上では、明確に区別して理解することが重要である。

文字 (Character)

文字とは、人間が言語を表現するために使用する記号や図形のことを指す。例えば、「あ」「い」「う」「A」「B」「C」「1」「2」「3」「漢」「字」などは、すべて文字である。文字は、抽象的な概念であり、それ自体はコンピュータで直接扱うことはできない。

符号 (Code)

符号とは、情報を表現・伝達・蓄積するために用いられる記号や信号の体系を指す。文字をコンピュータで扱うためには、文字を特定の符号に変換（符号化）する必要がある。例えば、モールス符号では、「・」と「—」の組み合わせで文字や数字を表現する。コンピュータの世界では、一般的に**2進数**（0と1の組み合わせ）が用いられる。

文字コード (Character Code)

文字コードとは、文字や記号をコンピュータで扱うために、各文字や記号に割り当てられた固有の数値（符号）の体系、またはその数値自体を指す。文字コードを定めることで、異なるコンピュータやソフトウェア間でも、文字情報をやり取りすることが可能となる。

文字集合と符号化文字集合

文字コードを理解する上で重要な概念として、**文字集合**と**符号化文字集合**がある。

- **文字集合 (Character Set)**：文字コードで表現する対象となる文字や記号の集まりを指す。例えば、「JIS X 0208」は、日本語の文字や記号を集めた文字集合である。
- **符号化文字集合 (Coded Character Set)**：文字集合に含まれる各文字に対して、固有の数値（コードポイント）を割り当てたものを指す。例えば、「Unicode」は全世界の文字を収録することを目指した符号化文字集合である。
- **文字レパートリー (Character Repertoire)**：特定の文字コードで表現できる文字の集合のこと。文字集合とほぼ同義だが、より抽象的な概念として用いられることがある。

文字符号化方式 (Character Encoding Scheme)

文字符号化方式（または文字エンコーディング）とは、符号化文字集合で定められた数値（コードポイント）を、実際にコンピュータが扱うデータ（バイト列）として表現するための具体的な変換規則を指す。例えば、Unicodeの符号化文字集合に対しては、「UTF-8」「UTF-16」「UTF-32」などの複数の文字符号化方式が存在する。 Shift_JIS や EUC-JP は、JIS X 0208 などの文字集合を基に定義された文字符号化方式である。

字体・字形・文字包摂・縮退

漢字や、それを崩した変体仮名においては、アルファベットと比べて字形のバリエーションが極めて豊富である。特に、氏名においては異体字や旧字体、誤字俗字などが本人のアイデンティティとして通用している実態がある。住所においても旧字体や略字、表記揺れが見られる。商号においても、屋号や商標が文字として用いられる場合があるなど、字形の多様性は顕著である。このような背景から、文字を符号化する際に**文字包摂 (Character Unification)**という概念が重要となる。

字体とは、文字の抽象的な骨組みであり、個々の字形に共通する特徴を指す。言い換えれば、文字の設計における共通のルールであると言える。一方、**字形**とは、この字体が実際に表現された具体的な形状を指す。手書き文字、印刷文字など、実際に目に見える文字の形は全て字形である。

文字包摂とは、このように異なる字形であっても、それらを区別せず、同一の文字として扱うという考え方である。そして、異なる字形同士が同一の文字であるかどうかを判断する行為を**文字同定**と呼ぶ。漢字における多様な字形を、この字体の共通性を基準として同一の文字として扱うことによって、正確かつ効率的な情報処理が可能となる。

一方、**縮退 (Character Substitution)**とは、異なるシステム間で円滑に情報を受け渡すために、ある文字を、それに似た字形を持つ別の文字で置き換える処理を指す。これは特に、異なる文字集合間での変換を行う際に発生する現象である。氏名や商号で用いられる特殊な文字が、互換性の低いシステム間でやり取りされる際に縮退が起こることがある。

例えば、Unicodeに存在する「髙」（はしごだか）という文字を、Shift_JISという文字コード体系に変換する場合を考えてみよう。Shift_JISには、「髙」に相当する文字コードが存在しない。このような場合に、「髙」は、Shift_JISに存在する類似の字形を持つ文字、例えば「高」に置き換えられることがある。商号登記において、使用できない文字を類似の文字で代替するのも、広義の縮退と言えるだろう。このように、文字包摂が同一の文字としての扱いを定める一方で、縮退は異なる文字コード間での代替処理を指すという違いがある。

文字フォント (Character Font)

文字フォント（または単にフォント）とは、文字を画面や印刷物などに表示・出力する際の具体的な字形データ（デザイン）を指す。例えば、「MS ゴシック」「MS 明朝」「Times New Roman」「Arial」などは、すべてフォントである。文字コードはあくまでも文字に割り当てられた数値であり、フォントはその数値を基に具体的な字形を表現するために用いられる。

フォントデータの形式には大きく分けて、**ラスタフォント**と**アウトラインフォント**の2種類がある。

- **ラスタフォント (Raster Font)**：文字の形を点の集まり（ビットマップ）として表現するフォント形式。ドットフォントとも呼ばれる。拡大縮小するとギザギザ（ジャギー）が目立つ。データサイズは比較的小さいが、特定のサイズに最適化されているため、サイズごとにデータを用意する必要がある。

- **アウトラインフォント (Outline Font)**：文字の形を輪郭線（アウトライン）の座標データとして表現するフォント形式。ベクターフォントとも呼ばれる。拡大縮小しても滑らかに表示される。データサイズはラスタフォントより大きいが、一つのデータで様々なサイズに対応できる。代表的な形式として、Type1、TrueType、OpenTypeなどがある。

  - **Type1フォント**：Adobe Systemsが開発したアウトラインフォント形式。PostScript言語をベースにしており、主にプロフェッショナルな印刷業界で利用されてきた。Type1フォントでは、文字のアウトラインを**ベジェ曲線**を用いて表現する。ベジェ曲線は、始点、終点、および2つの制御点によって定義される滑らかな曲線であり、複雑な形状を効率的に表現できる。

- **TrueTypeフォント**：AppleとMicrosoftが共同で開発したアウトラインフォント形式。Type1フォントに対抗して開発され、WindowsやmacOSなどのOSに標準搭載されている。TrueTypeフォントでは、文字のアウトラインを**Bスプライン曲線**を用いて表現する。Bスプライン曲線は、始点、終点、および1つの制御点によって定義される曲線であり、Type1フォントよりも計算量が少なく、高速な描画が可能である。
- **OpenTypeフォント**：MicrosoftとAdobeが共同で開発したアウトラインフォント形式。TrueTypeフォントをベースに、より高度な機能（異体字、合字など）をサポートしている。OpenTypeフォントでは、Type1フォントとTrueTypeフォントの両方の技術を統合しており、**ベジェ曲線**と**Bスプライン曲線**の両方を使用できる。これにより、より柔軟で表現力豊かなフォントデザインが可能となっている。

組み文字 (Ligature)

組み文字とは、複数の文字を組み合わせて一つの文字のように見せる技法のこと。縦書きや横書きで、数字や単位などをバランスよく配置するために用いられる。

日本における事例としては、「㍻」（平成）、「℡」（電話）、「㍊」（ミリバール）などのように、複数の文字が組み合わさって1つの文字コードが割り当てられているものがある。また、「〱」のような**踊り字**、嵯峨本などの古活字で複数の崩し字をまとめて扱う際に使われた**連綿活字**も組み文字の一種と言える。例えば、日本語フォントには下記のような組み文字が登録されている。

①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳ⓐⓑⓒⓓⓔⓕⓖⓗⓘⓙⓚⓛⓜⓝⓞⓟⓠⓡⓢⓣⓤ
ⓥⓦⓧⓨⓩ⓫⓬⓭⓮⓯⓰⓱⓲⓳⓴⓵⓶⓷⓸⓹⓺⓻⓼⓽⓾㈠㈡㈢㈣㈤㈥㈦㈧㈨㈩㈪㈫㈬㈭㈮㈯
㈰㈱㈲㈳㈴㈵㈶㈷㈸㈹㈺㈻㈼㈽㈾㈿㉀㉁㉂㉃㉑㉒㉓㉔㉕㉖㉗㉘㉙㉚㉛㉜㉝㉞㉟㊀㊁㊂㊃㊄㊅
㊆㊇㊈㊉㊊㊋㊌㊍㊎㊏㊐㊑㊒㊓㊔㊕㊖㊗㊘㊙㊚㊛㊜㊝㊞㊟㊠㊡㊢㊣㊤㊥㊦㊧㊨㊩㊪㊫㊬㊭㊮
㊯㊰㊱㊲㊳㊴㊵㊶㊷㊸㊹㊺㊻㊼㊽㊾㊿㋐㋑㋒㋓㋔㋕㋖㋗㋘㋙㋚㋛㋜㋝㋞㋟㋠㋡㋢㋣㋤㋥㋦㋧
㋨㋩㋪㋫㋬㋭㋮㋯㋰㋱㋲㋳㋴㋵㋶㋷㋸㋹㋺㋻㋼㋽㋾㋿㌀㌁㌂㌃㌄㌅㌆㌇㌈㌉㌊㌋㌌㌍㌎㌏㌐
㌑㌒㌓㌔㌕㌖㌗㌘㌙㌚㌛㌜㌝㌞㌟㌠㌡㌢㌣㌤㌥㌦㌧㌨㌩㌪㌫㌬㌭㌮㌯㌰㌱㌲㌳㌴㌵㌶㌷㌸㌹㌺
㌻㌼㌽㌾㌿㍀㍁㍂㍃㍄㍅㍆㍇㍈㍉㍊㍋㍌㍍㍎㍏㍐㍑㍒㍓㍔㍕㍖㍗㍱㍻㍼㍽㍾㍿㎅㎆㎇㎈㎉㎍
㎎㎏㎐㎖㎗㎘㎛㎜㎝㎞㎟㎠㎡㎢㎣㎤㎥㎦㎰㎱㎲㎳㏂㏄㏈㏋㏍㏔㏗㏘㏚

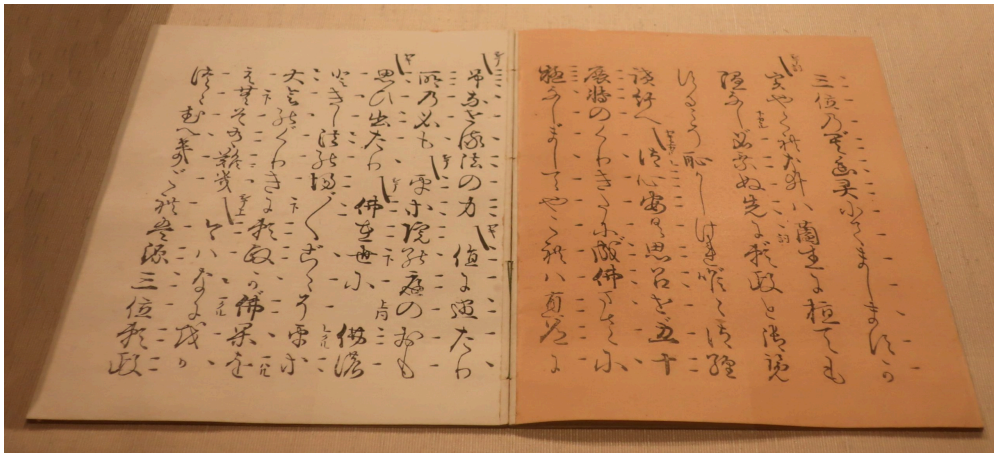


図1: 嵯峨本：能楽「葛城」の台本、本阿弥光悦作、江戸時代、1615年頃、木版印刷本

海外における事例としては、アルファベットの**合字（リガチャ）**が挙げられる。例えば、"f"と"i"を組み合わせた"fi"、"f"と"l"を組み合わせた"fl"などがある。これらは、"fi"、"fl"と連続する並びで文字間が詰まりすぎるのを防ぎ、見た目を美しくする目的でデザインされたものである。"f"と"f"と"i"を組み合わせた"ffi"、"f"と"f"と"l"を組み合わせた"ffl"といった、3文字以上の組み合わせも存在する。これらの合字の一部には、Unicodeにおいて独立したコードポイントが割り当てられている。

組み文字の技術は、単に文字コードの機能として存在するだけでなく、活版印刷の時代からその歴史を持っている。活版印刷においては、物理的な活字を組み合わせることで、複数の文字を一体として表現する技術が用いられていた。これにより、特定の文字の組み合わせを効率的に印刷したり、デザイン的な調整を行ったりすることが可能となっていた。

結合文字 (Combining Characters)

結合文字とは、他の文字と組み合わせて、アクセント記号や発音記号などを表現するために用いられる特殊な文字のことである。例えば、日本語の「が」は「か」と濁点、「ぱ」は「は」と半濁点を組み合わせて表現される。日本語の濁点・半濁点もUnicodeにおいて結合文字として定義されており、濁点はU+3099、半濁点はU+309Aのコードポイントが割り当てられている。これらは基底文字と組み合わせることで一つの文字として表現され、macOSのファイルシステムなどでも実際に使用されている。

国際的には、ラテン文字におけるアクセント記号（「é」のような鋭アクセント）や分音記号（「naïve」の「ï」）、タイ語やデーバナーガリー文字における母音記号や声調記号、アラビア語における文字の位置に応じた形状表現など、幅広い用途で使用されている。Unicodeでは、主要な結合文字のために「U+0300」から「U+036F」の範囲に専用のコードブロックが用意されており、これらの文字は単独では意味を持たず、必ず基底文字と組み合わせて使用される仕組みとなっている。

組み文字と結合文字の違い

組み文字（合字、リガチャ）と**結合文字**は、どちらも複数の文字を組み合わせて表示する技術であるが、その目的と仕組みが異なる。

- **組み文字**（合字、リガチャ）は、特定の文字の組み合わせに対して、デザイン上の理由から一つの図形として表現するものである。例えば、"fi" を "fi" のように表示する。これらは、見た目を美しくしたり、読みやすくするために使われる。

- **結合文字**は、複数のコードポイントを組み合わせて一つの文字を表現する仕組みである。例えば、ラテン文字にアクセント記号を付加する場合や、日本語の濁点や半濁点を付加する場合に使用される。結合文字は、文字の見た目を変更するために使われる。

つまり、組み文字は主に見た目の調整を目的とするのに対し、結合文字は文字の構成要素を組み合わせるための仕組みであると言える。

その他の考慮事項

制御文字 (Control Characters)

制御文字とは、文字コードにおいて、文字の表示やテキストの処理を制御するために用いられる特殊な文字のこと。例えば、改行（LF, CR）、タブ（TAB）、エスケープ（ESC）などが制御文字に該当する。これらの文字は、画面に文字として表示されるのではなく、テキストのレイアウトや通信プロトコルなどで特別な意味を持つ。

バイトオーダーマーク (BOM)

バイトオーダーマーク（BOM: Byte Order Mark）とは、Unicodeのテキストファイルにおいて、エンディアン（バイト順）を示すためにファイルの先頭に付加される特殊な符号のことである。UTF-16やUTF-32などのマルチバイト文字コードでは、バイトの並び順（エンディアン）が重要となるため、BOMによって正しく解釈する必要がある。

- **ビッグエンディアン (Big-Endian)**：複数バイトで構成されるデータを、上位のバイトから順に並べる方式である。例えば、2バイトのデータ「0x1234」は、メモリ上では「0x12 0x34」のように格納される。
- **リトルエンディアン (Little-Endian)**：複数バイトで構成されるデータを、下位のバイトから順に並べる方式である。例えば、2バイトのデータ「0x1234」は、メモリ上では「0x34 0x12」のように格納される。

BOMは、テキストファイルの先頭に特定のバイト列を付加することで、そのファイルがビッグエンディアンでエンコードされているか、リトルエンディアンでエンコードされているかを識別するために使用される。

文字コードの検出

文字コードの検出とは、テキストファイルやデータストリームの文字コードを自動的に判別する処理のこと。文字コードが不明な場合、誤った文字コードで解釈すると文字化けが発生するため、文字コードの検出は重要な技術となる。

文字コード検出の技法

文字コードを検出するための代表的な技法を以下に示す。

- **BOM (Byte Order Mark) の確認:** ファイルの先頭に付与されるBOMを調べることで、UTF-8、UTF-16 (LE, BE)、UTF-32 (LE, BE) といったUnicode系のエンコーディングを高精度に識別できる。BOMは、特にUTF-16やUTF-32のようにバイト順序が重要なエンコーディングにおいて有効である。

- **頻出バイト列のパターンマッチング:** 各文字コードには、特定の文字が特定のバイト列で表現されるという特徴がある。この特徴を利用し、ファイル中に頻出するバイト列のパターンを分析することで、文字コードを特定する。例えば、UTF-8でよく使われるマルチバイト文字のパターンや、Shift_JISでよく使われる2バイト文字のパターンなどを照合する。

- **統計的手法:** テキストデータにおける文字の出現頻度は、言語や文字コードによって異なる傾向がある。この統計的な情報を利用し、テキスト中の文字の出現頻度を分析し、既知の文字コードの統計データと比較することで、可能性の高い文字コードを推測する。

- **言語モデルの利用:** テキストの内容を解析し、その言語で一般的に使用される文字コードを推測する。例えば、日本語のテキストであれば、UTF-8、Shift_JIS、EUC-JP などが候補となる。より高度な手法では、機械学習を用いてテキストの内容から言語を特定し、その言語に最適な文字コードを推測する。

必ずしも適切に自動検出が働かない事例

上記の文字コード検出の技法は強力であるが、全ての場合において正確に動作するとは限らない。以下に、文字コードの検出が困難となる事例を示す。

- **BOM が存在しない場合:** 特に UTF-8 は BOM なしで広く利用されているため、BOM による判別ができない場合がある。この場合、他の検出方法に頼る必要があるが、精度が低下する可能性がある。

- **短すぎるテキストや特定の文字しか含まれないテキスト:** 文字コードの検出は、テキストに含まれる情報量に依存する。短いテキストや、ASCII 文字など、複数の文字コードで共通する文字しか含まれないテキストでは、パターンマッチングや統計的手法による判別が困難となる。

- **複数のエンコーディングで有効なバイト列:** 例えば、ASCII でエンコードされたテキストは、UTF-8 のテキストとしても有効である。このような場合、UTF-8 として誤って検出される可能性がある。

- **非テキストファイルの場合:** 画像ファイルや実行ファイルなど、テキストデータではないファイルを文字コード解析しようとしても、意味のある結果は得られない。これらのファイルは特定の構造を持っており、文字コード検出アルゴリズムを混乱させる可能性がある。

- **不適切なメタデータによる文字化け:** Content-Type header や XML 宣言などに記載されている文字コードに関するメタデータが、データの実体と違っていたり、記述が矛盾している場合など、文字化けが発生する可能性がある。

- **混合エンコーディング:** 同一ファイル内に複数の文字エンコーディングが混在している場合、正確な検出が困難となる。特にレガシーシステムとの連携時に発生しやすい問題である。

- **特殊な制御文字の存在:** 制御文字やエスケープシーケンスが含まれる場合、文字コード判定のアルゴリズムが誤動作する可能性がある。これは特にレガシーシステムや特殊な用途のテキストファイルで発生することがある。

文字の成り立ち

本章では、現代の日本語を記述する上で不可欠な漢字、平仮名、カタカナという三種類の文字の成り立ちを、人類における文字の誕生、そして漢字の故郷である中国における発展というより広い視点から捉え直す。特に、漢字の多様な書体の成立と発展、そして印刷技術における活字の歴史に焦点を当て、印刷や書道における役割を強調する。さらに、漢字の整理と知識の伝達に不可欠な漢字辞書の調製についても概説し、文字コードを学ぶ上で不可欠な基礎知識を提供する。

人類の文字の起源：記録と伝達への渇望

文字は、人類が情報を記録し、遠くへ伝え、未来へと残すための最も重要な発明の一つである。その起源は一様ではないが、紀元前3000年頃のメソポタミア文明における楔形文字や、エジプト文明におけるヒエログリフなどが最古の文字体系として知られている。これらの初期の文字は、具体的な事物や概念を表す表語文字としての性格が強かった。

図1: セティ1世（在位紀元前13世紀）の墓から発見された象形文字が刻まれた壁の断片

その後、文字はそれぞれの文化や言語に合わせて多様な発展を遂げる。音と文字を結びつける試みが始まり、音節文字や、さらに音素を表すアルファベットが誕生する。アルファベットは少数の文字で多様な音を表現できるため、世界各地へと広がり、現代の多くの言語で使用されている。

中国における漢字の誕生と発展

日本語の文字を理解する上で欠かせないのが、漢字の存在である。漢字は、数千年の歴史を持つ中国で生まれた表語文字であり、その起源は紀元前14世紀頃の殷王朝時代に用いられた甲骨文字に遡ると考えられている。甲骨文字は、亀の甲羅や獣骨に刻まれた占いの記録であり、象形文字としての特徴を色濃く残している。

その後、漢字は青銅器に鋳込まれた金文へと発展する。金文は、甲骨文字よりも整った形状を持ち、文字の種類も増加している。春秋戦国時代に入ると、地域によって字形が異なる状態が生まれたが、秦の始皇帝による中国統一を経て、文字の統一が行われ、**篆書**(てんしょ)が標準書体となる。篆書は、曲線的で左右対称な形状が特徴であり、現代でも印章などに用いられている。

図2：秦権　廿六年詔権量銘全文（「始皇七刻石」とともに数少ない篆書体の書蹟）

篆書を簡略化し、より実用的にしたのが**隷書**(れいしょ)である。隷書は、秦代に官吏が日常業務で使用するために生まれた書体で、波打つような筆画（波磔）が特徴である。隷書の成立は、漢字の書体の大きな転換点となり、後の様々な書体の基礎となった。

後漢時代に入ると、隷書からさらに発展する形で、現代の漢字の基本となる**楷書**(かいしょ)が成立する。楷書は、点画を一つ一つ明確に書くことを特徴とし、整然とした美しい字形を持つ。楷書は、規範的な書体として広く用いられ、印刷書体である明朝体の基礎ともなった。

楷書とほぼ同時期に、隷書を速記するために生まれたのが**草書**(そうしょ)である。草書は、筆画を省略したり、続けたりすることで、流れるような動きのある字形を生み出す。草書には、隷書の面影を残す**章草**(しょうそう)、より自由で奔放な**今草**(こんそう)、そして極端に筆画を省略した**狂草**(きょうそう)といった種類がある。草書

は、書道における重要な表現の一つであり、日本の平仮名の成立にも大きな影響を与えた。また、楷書をやや崩して書きやすくした**行書**も広く用いられるようになった。行書は、楷書の可読性と草書の速記性を兼ね備えており、日常的な筆記に広く用いられる。

膠泥活字の発明と明朝体

漢字の歴史において特筆すべきは、印刷技術における活字の発明である。中国では、11世紀の北宋時代に**畢昇**によって**膠泥活字**、すなわち陶製の活字が発明されたとされている。これは、個々の文字を独立した部品として作成し、必要に応じて組み合わせて印刷を行う画期的な技術であった。その後、木製の活字や金属製の活字も用いられるようになり、特に明の時代には、青銅活字を用いた大規模な印刷が行われた。これらの活字技術は、書籍の大量印刷を可能にし、知識の普及に大きく貢献した。特に、印刷に適した書体として、直線的な筆画を持ち、視認性の高い**宋朝体**が発展し、これが後の**明朝体**へと繋がっていく。明朝体は、横線が細く、縦線が太いという特徴的な構造を持ち、筆画の端には「うろこ」と呼ばれる三角形の装飾が見られる。明朝体は、可読性が高く、印刷物に最適な書体として、現代においても広く用いられている。

図3: 膠泥活字で印刷された世界最古の活版印刷物「佛说观无量寿佛经」温州博物馆藏

漢字辞書の編纂

漢字は数が多い上に、一つの漢字が複数の読みや意味を持つため、漢字を効率的に理解し、活用するためには、漢字を整理し、検索するための道具、すなわち漢字辞書が不可欠となる。中国においては、後漢時代の許慎によって編纂された『説文解字』が、漢字の体系的な分析と字源解説を行った最初の辞書として知られている。そして、清朝の康熙帝の勅命により編纂された『**康熙字典**』は、1716年に完成し、部首によって漢字を分類し、字音、字義、用例などを詳細に解説した大規模な辞書である。その収録文字数は約47,000字に及び、長らく漢字研究の基礎資料として、中国のみならず日本を含む東アジア地域で重要な役割を果たしてきた。

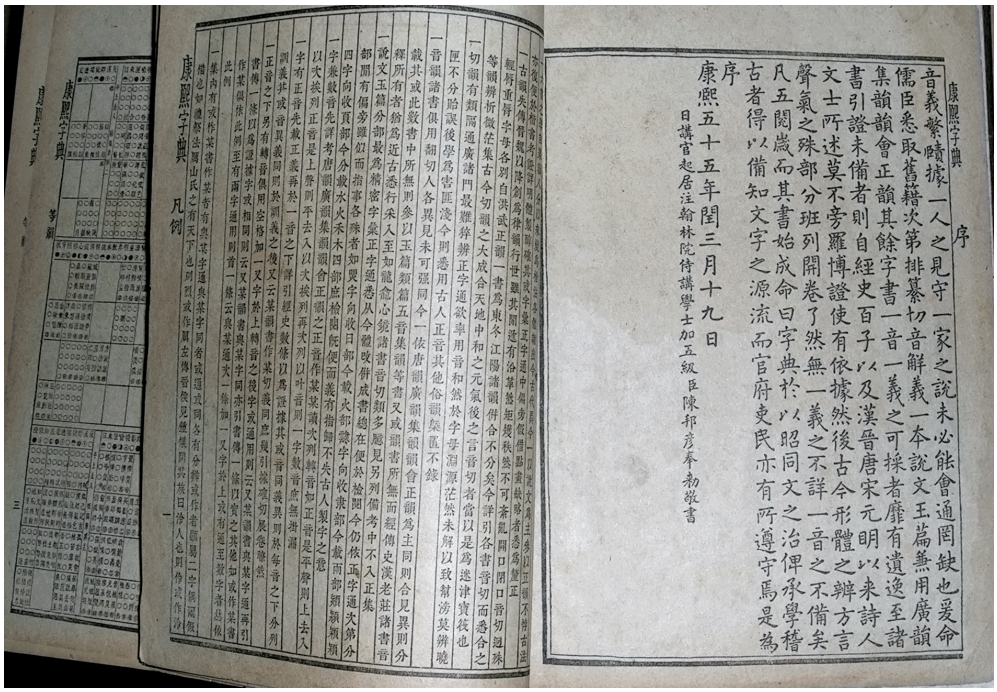


図4：康熙字典

日本における最古の漢字辞書としては、平安時代に成立した『倭名類聚抄』が挙げられる。これは、漢字の意味に基づいて分類された類書であり、現代の漢字辞書とは形式が異なるものの、当時の漢字知識を知る上で重要な資料である。

江戸時代に入ると、印刷技術の発展や教育の普及に伴い、体系的な漢字辞書が編纂されるようになる。貝原益軒の『和訓栞』や、新井白石の『同文通考』などがその例である。これらの辞書は、漢字の音訓、意味、用例などを詳細に解説しており、漢字学習に大きく貢献した。

明治時代以降は、西洋の辞書学の影響を受け、音訓索引や部首索引など、より効率的な検索方法を取り入れた近代的な漢字辞書が主流となる。大槻文彦の『言海』や諸橋轍次の『大漢和辞典』などは、近代日本の代表的な漢字辞書として知られている。

漢字の伝来と日本語への適応

日本に漢字が伝来したのは、明確な時期は断定できないものの、古墳時代には既に朝鮮半島を経由して伝わっていたと考えられている。当初、漢字は主として政治や宗教に関わる知識層によって使用され、日本語の固有名詞や意味を表すために用いられた。この際、楷書体が中心に伝えられたと考えられる。

しかし、日本語の音韻構造は漢字と大きく異なるため、漢字をそのまま日本語の表記に用いるには限界があった。そこで考案されたのが、漢字の音を借りて日本語の音を表す「万葉仮名」である。これは、特定の漢字をその意味に関わらず、音を表す記号として用いる方法であり、現存する最古の歌集『万葉集』にその例を見ることができる。万葉仮名の使用は、後の平仮名やカタカナの成立に重要な影響を与えることとなる。特に、草書体が平仮名の成立に大きな役割を果たした。

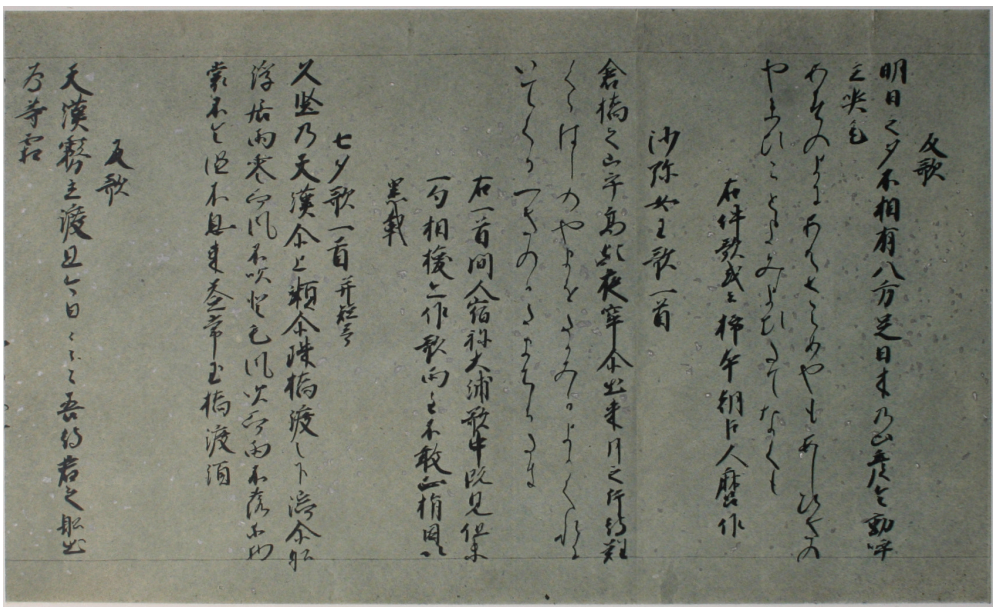


図5：藍紙本万葉集　第9巻　平安後期　（京都国立博物館）

平仮名とカタカナの誕生

平安時代に入ると、万葉仮名を簡略化する形で、二種類の表音文字、すなわち平仮名とカタカナが誕生する。平仮名は、万葉仮名を草書体で書き崩したものが起源とされる。この過程で、特定の漢字に固定された字形だけでなく、同じ音を表す様々な漢字の草書体が用いられるようになり、これらは**変体仮名**（へんたいがな）と呼ばれる。変体仮名は、平仮名の一種でありながら、その字形は多岐にわたり、書き手の個性や時代による字形の変化を反映していた。和歌や物語など、主に女性の間で用いられ、優美で流麗な表現を可能にした。当初、平仮名は現代の五十音に収束しておらず、同じ音に対して複数の変体仮名が存在するのが一般的であった。例えば、「あ」の音に対しては「安」「阿」「悪」など様々な漢字を草書体化した変体仮名が用いられた。

一方、カタカナは、漢字の一部を取って作られたものが起源とされる。直線的で角ばった印象を持つカタカナは、主に仏教経典の訓点や注釈を書き記すために、僧侶の間で用いられた。カタカナは、平仮名のような変体仮名の発達は見られず、比較的早い段階から字形が固定化された。

明治時代に入ると、教育の普及や印刷技術の発展に伴い、文字の標準化が強く求められるようになる。その結果、平仮名についても、同じ音を表す複数の変体仮名の中から代表的なものが選ばれ、整理統合が進められた。1900年（明治33年）に小学校令施行規則が改正され、現在の五十音に基づく平仮名が正式なものとして定められ、変体仮名は日常的な使用から徐々に姿を消していった。しかし、変体仮名は、書道や古典籍、看板など、特定の分野においては現在でもその姿を見ることができる。このように、平仮名は変体仮名という多様な形態を経て、現代の五十音に収束した。

図6：当時の国語教科書における綴字篇第12課

多様な文字体系と印刷技術の礎

人類は、それぞれの文化や言語に合わせて多様な文字体系を生み出してきた。表語文字、音節文字、アルファベット、そして漢字、平仮名、カタカナといった文字は、それぞれが異なる歴史と特性を持っている。特に漢字は、その長い歴史の中で多様な書体を生み出し、それぞれの書体が文化や技術の発展と深く関わってきた。中国における活字の発明は、漢字文化の普及に決定的な役割を果たし、現代の印刷技術の礎となっている。

現代社会において、これらの多様な文字をコンピュータ上で処理し、世界中の人々と情報をやり取りするためには、文字を数値化し、共通の規格で扱う必要がある。これが文字コードの役割である。

この序章で見てきたように、日本語の文字は、人類の文字の歴史、そして漢字という独自の文字体系、特にその多様な書体と活字の歴史の影響を受けながら、独自の発展を遂げてきた。この複雑な成り立ちを理解することは、文字コードの本質を理解するための第一歩となる。次章からは、主要な文字集合と、文字コード形成の歴史について説明する。

主要な漢字辞書の概要

辞書名	成立年代	編者	収録字数	主な特徴
説文解字	100年頃	許慎	約9,353字	部首別配列、字源解説を重視した中国最古の体系的な字書
倭名類聚抄	931年頃	源順	漢字約3千字	日本における最古の部類辞書で、漢語に和訓を付す
康熙字典	1716年	張玉書ら	約47千字	部首別配列、字音、字義、用例を解説した大規模な辞書
和訓栞	1777年	貝原益軒	約8千字	字音、字義、和訓、用例を詳細に解説
同文通考	1811年	新井白石	不明	字音、字義、和訓に加え、中国・朝鮮の音韻も参照
言海	1891年	大槻文彦	約5万語	日本語の辞書だが多くの漢字項目を含む。字源解説も行う
大漢和辞典	1960年	諸橋轍次	約5万字	漢字の字音、字義、字源、用例を網羅的に収録した、日本最大規模の漢和辞典

文字集合の歴史

情報化社会の基盤をなす文字コードは、単に文字をデジタル化するための技術的な枠組みにとどまらず、その国の文化、歴史、そして社会構造を映し出す鏡とも言える。文字コードは、対象とする文字の集まりである文字集合を定めた上で、その文字集合に含まれる各文字に固有の数値（符号）を割り当てることで成り立っている。特に、日本語のように多種多様な文字種（ひらがな、カタカナ、漢字）を用いる言語において、どのような文字集合を採用するかは、単なる技術的な選択を超えた、極めて重要な意味を持つ。

例えば、パソコンやスマートフォンで当たり前のように使われている漢字の背後には、実は長い試行錯誤と議論の歴史が隠されている。なぜ常用漢字は2,136字なのか？なぜ人名に使える漢字は特別に定められているのか？そして、それらの漢字はどのようにして選ばれてきたのか？これらの疑問は、日本の近代化の歩みと、その中で繰り広げられてきた「国語国字問題」を抜きにしては語れない。

本章では、行政事務で扱う上で特に重要な文字集合に焦点を当て、その歴史的変遷を丁寧に紐解いていく。具体的には、当用漢字、常用漢字、そして人名用漢字がどのように制定され、現代社会にどのような影響を与えてきたのかを詳細に検討する。さらに、日本における漢字制限の歴史を、明治期の国語国字改革運動から説き起こし、印刷技術の発展や社会情勢の変化との関連性にも注目しながら、多角的に考察していく。この歴史的な視座は、単に過去の経緯を知るためだけではなく、私たちが未来の文字文化をどのように築いていくべきか、その指針を得るためにも不可欠なものとなるだろう。

日本語の文字と印刷技術

日本の国語国字問題の起源は、16世紀末まで遡ることができる。1579年に来日したイエズス会の東インド巡察師アレッサンドロ・ヴァリニャーノは、日本語の文字の複雑さが、ヨーロッパ人宣教師と日本人信徒間の円滑なコミュニケーションを妨げ、布教活動の大きな障壁となっていると認識した。彼は既に1583年には、日本語の印刷物は膨大な数の漢字を必要とし、その印刷は困難を極めるため、ローマ字表記を採用することが妥当であると主張していた。この問題意識は、後の明治期における漢字制限論と本質的に変わらないものであった。

明治維新以降、日本の急速な近代化に伴い、国語国字問題は再び大きな議論の的となった。特に、漢字の多さと複雑さが教育や情報伝達の効率化を阻害しているとの考えから、漢字の制限や廃止、そして仮名文字やローマ字の採用が提唱された。この議論は、日本語の活版印刷技術の歴史とも密接に関連していた。

日本における活版印刷の歴史は16世紀末に遡り、西洋と朝鮮という二つの異なるルートから技術が伝来した。西洋からはイエズス会の宣教師によってグーテンベルク式の活版印刷機が、朝鮮からは文禄・慶長の役を通じて朝鮮式の活版印刷技術がもたらされた。初期のキリシタン版では、日本語の文字に関する課題への実践的な解決策として、ローマ字版と漢字仮名交じり文版の両方を制作するというアプローチが採用された。これは、宣教師と日本人信徒、双方の言語理解力に対応するため、そして活字製作の効率化を考慮した上での選択であった。しかし、禁教令によってこの印刷活動は中断を余儀なくされ、江戸時代には主に木版印刷が主流となった。

図1：16世紀のキリシタン版（印刷博物館所蔵）

明治期に入り、活版印刷は再び本格的に普及し始めたが、漢字の多さは依然として大きな課題であった。活版印刷には数千種類もの漢字活字が必要であり、さらに文字ごとに様々なサイズの活字が必要とされた。そのため、印刷会社は数万規模の膨大な活字を保有し、それらを正確かつ迅速に組み合わせて文選を行うためには、専門的な技能を持つ文選工や植字工が不可欠であった。この技術的な課題は、16世紀末にヴァリニャーノが指摘してから約300年を経てもなお、本質的には解決されていなかったのである。

図2: 日本語の活字は膨大な数になる

戦前における漢字制限の伏線

明治初期の日本語ローマ字化運動

明治維新によってもたらされた急速な社会変革は、当然ながら言語政策にも大きな影響を及ぼした。特に、西洋の進んだ学問や技術を積極的に吸収しようとする機運が高まる中、複雑な日本語の文字体系は、近代化を阻害する要因の一つと見なされるようになった。識字率の低さも問題視され、教育の普及と効率化の観点からも、国語国字の改革は避けて通れない課題となっていた。

こうした状況下で、日本語のローマ字化運動が一部の知識人らによって提唱された。この運動の嚆矢となったのは、1869年（明治2年）に前島密が将軍徳川慶喜に提出した「漢字御廃止之議」である。前島は、漢字の学習に要する多大な労力と時間が、国民の知識向上を妨げているとし、漢字を廃止して表音文字である仮名文字を採用すべきだと主張した。

この漢字廃止論は、当時としては急進的なものであったが、一定の支持を集め、1870年代には、ローマ字を用いて日本語を表記する「羅馬字会（らまじかい）」が結成されるなど、具体的な運動へと発展していった。羅馬字会の会員には、南部栄信、西周、森有礼といった、当時の日本を代表する知識人たちが名を連ね、ローマ字の普及活動を展開した。

羅馬字会は、ローマ字表記法の制定やローマ字による出版物の刊行などを通じて、その主張を広めていった。彼らの活動は、一般社会にも徐々に影響を与え、ローマ字による教科書や新聞が発行されるなど、一定の成果を上げた。

しかし、ローマ字化運動は、多くの反対意見にも直面した。特に、漢字を長年使用してきた知識層からは、日本語の伝統や文化を破壊するものとして、強い反発を受けた。また、同音異義語の多い日本語をローマ字で表記することの難しさや、視覚的な情報量の低下なども指摘され、ローマ字化運動は徐々に勢いを失っていった。

急進的なローマ字化運動が退潮に向かう一方で、より現実的なアプローチとして、教育現場における漢字の負担軽減が模索されるようになった。この流れが、教授漢字の制限という具体的な施策として結実することになる。

1900年: 小学校令と教授漢字の制限

1900年（明治33年）、小学校令施行規則が発布され、平仮名の五十音を確立するとともに、尋常小学校で教授する漢字を1,200字以内に制限することが定められた。これは、教育現場における漢字の負担を軽減し、初等教育の効率化を図るための措置であった。1904年（明治37年）には国定「尋常小学読本」（第1期国定教科書）が発行され、教科書で使われる漢字もこの制限に沿って選定された。尋常科で501字、高等科で355字、合計857字の「**教授漢字**」が制定された。

1923年 文部省「常用漢字表」発表と略字制定

漢字制限の議論は続き、1923年（大正12年）には、臨時国語調査会が「常用漢字表」を発表した。これは、日常使用する漢字を1,962字に制限し、さらに略字154字を定めたものであった。この常用漢字表は、当用漢字表制定以前における包括的な漢字制限案であり、当時の新聞・雑誌・出版業界の多くから支持を得ていた。また、同時に「仮名遣改定案」も提案され、これは後の「現代仮名遣い」の原型となるものであった。しかし、これらの改革案は、同年9月1日に発生した関東大震災により実施が見送られることとなった。その後、新聞社は独自の漢字選定を進めることとなり、大阪の新聞社が2,490字を、東京の新聞社が2,108字を選定した。

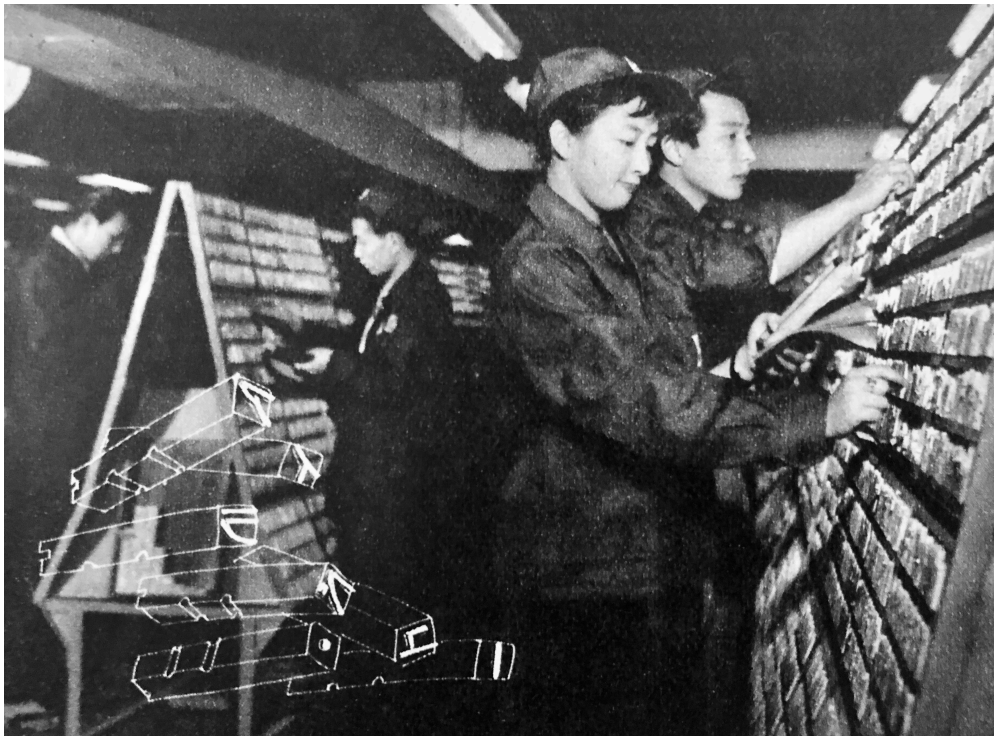


図3： 1955年3月頃。日本の新聞社の植字作業室　『サンケイグラフ』1955年3月27日号

1930年代の国語国字改革運動

1930年代に入ると、国粋主義や国家主義の台頭に伴い、国語国字改革運動はさらに活発化した。1931年（昭和6年）に石黒修（のちの厚生次官）を委員長とする国語問題研究会が「国語問題研究報告」を発表。国語問題研究会は陸軍関係者の支援を受け、漢字制限の必要性を訴えるとともに、「国語国字改良案」を提示し、1933年（昭和8年）には片仮名採用を文部大臣に建議した。しかし、この建議も反対派の抵抗により採択されなかった。

一方、文部省は1934年（昭和9年）に国語審議会を設置し、再び国語国字問題の審議を開始した。国語審議会では、漢字制限、仮名遣い、送仮名などについて議論が重ねられたが、具体的な成果は得られなかった。同時期に、陸軍は「兵器名称用制限漢字表」（1940年）を制定し、兵器の名称に使用する漢字を1,235字に制限する措置を講じた。

1942年 国語審議会「標準漢字表」答申

1942年（昭和17年）、国語審議会は「**標準漢字表**」を答申した。この表は、常用漢字1,134字、準常用漢字1,320字、特別漢字74字の合計2,528字からなり、従来の漢字制限案に比べて大幅に漢字数を削減したものであった。しかし、太平洋戦争の戦局悪化に伴い、この標準漢字表が実施されることはなかった。

戦前の漢字制限の挫折

戦前の漢字制限の試みは、国粋主義者や伝統主義者からの根強い反対、政府内の意見対立、そして戦争による社会情勢の変化など、様々な要因によって挫折を繰り返した。これらの試みが、のちの当用漢字表制定にどのような影響を与えたのか、また、戦時下における漢字制限がどのような意味を持っていたのかについては、今なお議論の余地がある。しかし、戦前の漢字制限に関する議論と実践は、日本の国語国字問題の歴史を理解する上で欠かせない。

戦後の当用漢字、人名用漢字、常用漢字

当用漢字

当用漢字とは、1946年（昭和21年）に内閣告示された「当用漢字表」で示された1,850字の漢字の集合である。法令、公用文書、新聞、雑誌、その他一般社会で使用する漢字の範囲を定めたもので、戦後の国語国字改革の一環として導入された。

当用漢字制定の背景と歴史

第二次世界大戦後、GHQは日本の民主化政策の一環として、国語国字問題に関与した。1946年3月に発表された「米国教育使節団報告書」では、日本の文字体系の複雑さが民主化の障害になっていると指摘され、ローマ字採用が勧告された。

GHQがローマ字採用を提案した背景には、複数の理由があった。まず、複雑な漢字体系が識字率を低下させ、それが民主化を遅らせる要因になると考えられていた。また、ローマ字化によって日本語の国際的な普及が進み、外国人の日本語学習も容易になると期待された。さらに、漢字学習に費やす時間を減らすことで、他の学習時間を確保できると考えられていた。

しかし、1948年に実施された「日本人の読み書き能力調査」で状況は大きく変わった。15歳から64歳までの約17,000人を対象とした調査の結果、漢字の読み書きができない人はわずか2.1%であることが判明したのである。この予想外に高い識字率により、GHQは完全なローマ字化政策を断念することとなった。

その代わりに、日本政府と協力して漸進的な文字改革を進めることになり、その一環として漢字使用の制限が検討された。国語審議会は1946年11月に「当用漢字表」（1,850字）を答申した。これは、完全なローマ字化ではなく、日本の文字体系を簡素化して教育の効率化を図りつつ、日本の文化的伝統も維持するという妥協策であった。

このように、当用漢字の制定は、GHQの意向と日本の実情との間で模索された改革の一つの到達点であったと言える。

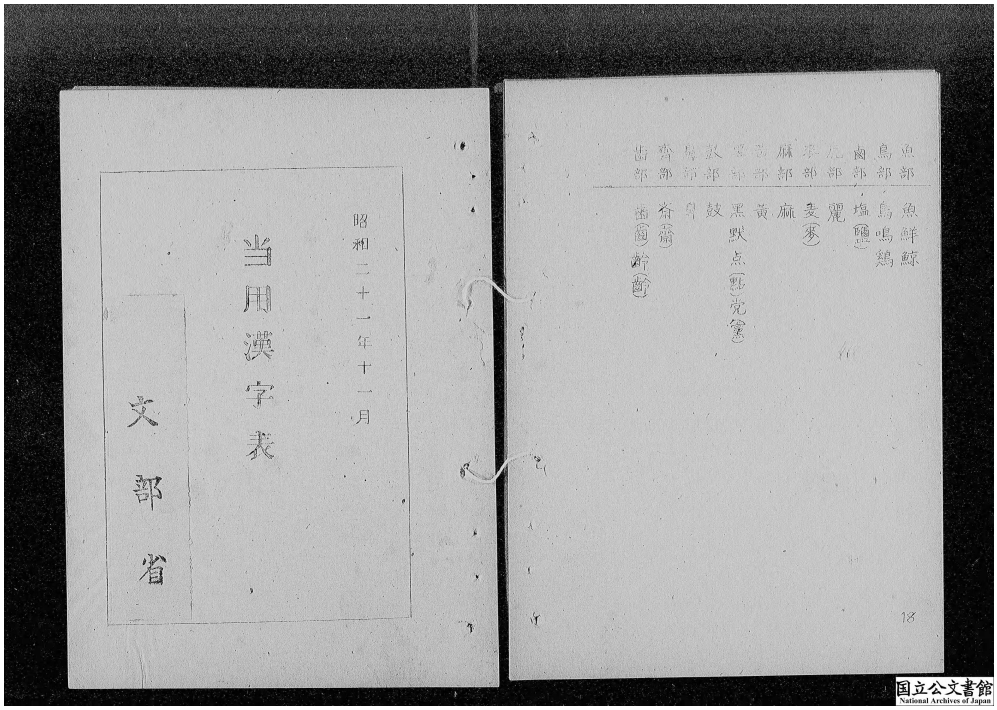


図4：当用漢字表 昭和21年11月（国立公文書館）

当用漢字の特徴

当用漢字表では、使用できる漢字を1,850字に制限した。これは、それまで一般的に使用されていた漢字数を大幅に削減するものであった。制限の対象となる漢字の選定は、日常生活における必要性や使用頻度を基準として行われた。ただし、選定された漢字の中には、現代ではあまり使用されない字も含まれていた。また、当用漢字表に含まれない漢字は、原則として公文書や教科書での使用が制限されることとなった。

当用漢字表と同時に「当用漢字字体表」が示され、多くの漢字で字体の簡略化が行われた。これは新字体と呼ばれ、漢字の学習負担を軽減し、表記の統一を図ることを目的としたものである。例えば「學」を「学」に、「國」を「国」に、「鐵」を「鉄」にするなど、画数を減らして書きやすくする工夫がなされた。この簡略化により、印刷や筆記の効率も向上することとなった。

「当用」という言葉には「さしあたり用いる」という意味が込められており、この制定は恒久的なものではなく、将来的な見直しの可能性を含んでいた。実際に、その後の社会変化や必要性に応じて、漢字政策は徐々に見直されることとなる。この暫定的な性格は、日本の言語政策における柔軟性を示すものであった。

当用漢字の影響と評価

当用漢字表は、法令、公用文書、新聞、雑誌など、一般社会における漢字使用の基準となり、戦後の日本語表記に大きな影響を与えた。また、教育現場でも当用漢字表に基づく漢字教育が行われ、漢字の習得が効率化された。

しかし、当用漢字表に対する批判も存在した。例えば、制限された漢字の数が少なすぎるとの意見や、新字体の採用に対する反対意見などがあった。また、当用漢字表に含まれない漢字を使用する際には、いわゆる「混ぜ書き」（「交ぜ書き」とも表記される。例：「改訂」を「改定」、「弗素」を「フッ素」）や、仮名書き、別の語句による言い換えを強いられたことも、当用漢字が普及する上での課題となった。

常用漢字表への移行

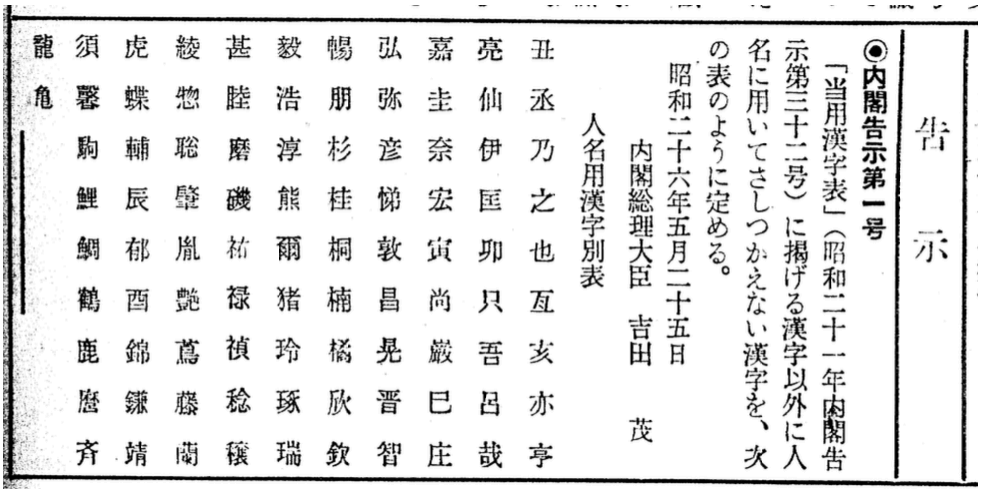
当用漢字表は、あくまでも「当用」の措置であり、恒久的なものではなかった。その後、社会の変化に伴い、当用漢字表の見直しが求められるようになった。そして、1981年（昭和56年）に「常用漢字表」が制定され、当用漢字表はその役割を終えた。常用漢字表では、当用漢字表に比べて漢字数が1,945字に拡大され、より実用的なものとなった。

現在では、公文書等で「当用漢字」という区分が用いられることはない。しかし、当用漢字表は、戦後の日本語表記の変遷を理解する上で、重要な役割を果たしたといえる。

人名用漢字

人名用漢字策定の背景

戦前は、子の名前に使用できる漢字に制限はなかった。しかし、当用漢字表の制定により、当用漢字表に含まれない漢字を子の名前に使用することができないという解釈が生じ、混乱を招いた。そこで、1948年（昭和23年）に戸籍法施行規則が改正され、当用漢字表以外の漢字で人名に用いても差し支えない漢字として、人名用漢字別表が定められた。



告示

◉内閣告示第一号

「当用漢字表」（昭和二十一年内閣告示第三十二号）に掲げる漢字以外に人名に用いてさしつかえない漢字を、次の表のように定める。

昭和二十六年五月二十五日

内閣総理大臣　吉田　茂

人名用漢字別表

丑 丞 乃 之 也 亙 亥 亦 亨
亮 仙 伊 匡 卯 只 吾 呂 哉
嘉 圭 奈 宏 寅 尚 巌 巳 庄
弘 弥 彦 悌 敦 昌 晃 晋 智
暢 朋 杉 桂 桐 楠 橘 欣 欽
毅 浩 淳 熊 爾 猪 玲 琢 瑞
甚 睦 磨 磯 祐 禄 禎 稔 穣
綾 惣 聡 肇 胤 嵐 艶 蔦 藤 蘭
虎 蝶 輔 辰 郁 酉 錦 鎌 靖
須 馨 駒 鯉 鯛 鶴 鹿 麿 斉
龍 亀

図5: 官報 第7310号 1951年5月25日 p.402 人名用漢字別表（官報告示）

新戸籍法施行以前の出生者への対応

1948年（昭和23年）の戸籍法施行規則改正および人名用漢字別表制定は、**それ以降に出生した者**に対して適用されるものであった。一方、**それ以前に出生した者**の名前については、以下の運用がなされた。

- **既に使用されている漢字**: 新戸籍法施行以前に既に名前に使用され、出生届が受理されていた漢字については、引き続き使用が認められた。
- **改名**: 人名用漢字別表に含まれない漢字が名前に使われている場合でも、必ずしも改名が強制されたわけではなかった。ただし、家庭裁判所の許可を得て、人名用漢字または常用漢字の範囲内で改名することは可能であった。
- **通称**: 正式な名前とは別に人名用漢字または常用漢字の範囲内で通称を使用することもできた。

このように新戸籍法施行以前に出生した者に関して、戸籍名と通称を使い分ける状況もあった。

人名に使える漢字の段階的な追加

人名用漢字は、社会情勢の変化や国民の要望などを踏まえ、適宜追加・変更が行われてきた。特に、2004年（平成16年）には、戸籍における命名に使用できる文字の見直しが行われ、人名用漢字が大幅に追加された。この変遷には、判例の影響も無視できない。

- **「曽(曾)」をめぐる訴訟（平成５年最高裁判決）**: 常用漢字の「曽」ではなく、当用漢字に含まれない旧字体の「曾」を子の名前に使用することが認められなかったことを不服として争われた訴訟。最高裁は、「曾」の字は、子の名に使用を認められる慣用字体であるとして、出生届を受理しなかった市側の決定を違法と判断した。この判決は、**人名用漢字表にない漢字であっても、社会通念上、人名に用いるのに不適切とは認められない漢字は、出生届を受理しうる**という解釈を示した。
- **「琉」をめぐる訴訟（平成15年最高裁判決）**: 「琉」の字は、2004年の戸籍法改正まで人名用漢字として認められていなかった。「琉」を含む名前の出生届が受理されなかったことを不服として争われた訴訟。最高裁は、**子の命名権の濫用に当たる場合を除き、戸籍法５０条１項違反を理由に出生届を受理しないことは許されない**という判断を示した。この判決は、命名権を広く認めるものであり、人名用漢字表の増補・改正に大きな影響を与えた。

これらの判例は、**人名用漢字表が絶対的な基準ではなく、社会通念や個別の事情を考慮する必要がある**ことを示した。その結果、人名用漢字は、より柔軟に運用されるようになり、国民のニーズに合わせた見直しが続けられている。

常用漢字

常用漢字は、1981年（昭和56年）に内閣告示された「常用漢字表」で示された漢字の集合である。常用漢字表は、当用漢字表に代わるものとして、法令、公用文書、新聞、雑誌、放送など、一般の社会生活において使用する漢字の目安を示したものである。

漢字制限から漢字活用への転換

当用漢字表は、戦後の国語国字改革の一環として、漢字使用を制限する目的で制定された。しかし、社会実態に合わない面もあり、以下のような批判が根強く存在した。

- **制限が厳しすぎ、表現に制約が生じる**: 当用漢字表に含まれない漢字の使用が制限されたため、固有名詞や専門用語などの表記に不便が生じた。また、文学作品などにおける表現の幅が狭められるとの批判もあった。

- **「混ぜ書き」の増加**: 当用漢字表にない漢字を含む語を仮名で表記する、いわゆる「混ぜ書き」が増加し、かえって読みにくさを招いた。
- **当用漢字表自体の形骸化**: 科学技術の発展や外来語の増加などにより、当用漢字表だけでは対応できない場面が増え、次第に当用漢字表自体が形骸化していった。

これらの批判を受け、国語審議会は、漢字制限という発想を転換し、**漢字と仮名を適切に使い分ける**という方向で検討を進めた。その結果、当用漢字表を抜本的に見直し、漢字使用の目安を示すものとして、常用漢字表が答申されたのである。

常用漢字の特徴

- **目安としての性格**: 常用漢字表は、当用漢字表と異なり、漢字使用を制限するものではなく、あくまでも目安として位置づけられた。「この表に掲げる漢字が、現代の国語を書き表す場合の漢字使用の目安となるように」と記され、制限ではなく目安であることが示された。
- **漢字数の増加**: 当用漢字表の1,850字から1,945字に拡大された（2010年の改定で2,136字）。
- **字体・音訓の整理**: 字体や音訓についても、より実態に即したものに改められた。
- **「付表」の存在**: 常用漢字表には、慣用が固定していると認められる表外漢字・表外音訓について、「付表」で示されている（2010年改定時に廃止され、「（参考）」欄に変更）。

韓国におけるハングル専用化との比較

東アジアの漢字文化圏において、20世紀中頃から文字改革の大きな動きが見られた。特に韓国における「ハングル専用化」の取り組みは、日本の漢字制限政策とは異なる特徴的な展開を見せた。

朝鮮時代、ハングルは支配層から「諺文」「女の文字」「子どもの文字」と呼ばれ、正統な文字とは見なされていなかった。公文書は漢文で記され、科挙試験でも漢文の能力が重視された。

日本統治時代（1910-1945年）には、学校教育で日本語が主要言語となり、公文書や新聞でも漢字とハングルの混用が一般的であった。この時期に多くの和製漢語や日本語発音の単語が朝鮮に導入され、解放後の言語政策に大きな影響を与えることとなった。

1948年、李承晩大統領は建国直後にハングル専用法を制定し、公文書のハングル表記を原則とした。当時の識字率は低く、漢字教育を受けた層は限られていたため、「当分の間、漢字を併用できる」という例外規定が設けられた。この政策は漢字使用を完全に禁止するのではなく、新たな漢字教育を制限することで、一世代かけてハングル専用世代を育成する戦略であった。

1970年には朴正熙政権下で本格的なハングル専用化の強化政策が実施された。教科書や公文書からハングル以外の文字が排除され、特に1970年から1972年に中高生だった世代は、漢字教育を全く受けず、自分の名前の漢字すら知らないケースも少なくなかった。

ハングル専用化政策は民族的アイデンティティの確立という目的では一定の成果を上げた。しかし、同音異義語の多発による読みづらさや、漢字を用いない場合の語の正確な意味理解の困難さなど、実用面での課題も浮上した。

1980年代以降、こうした課題に対応するため、中学校での漢字教育が段階的に再導入された。2005年には国語基本法が制定され、ハングルを標準としながらも必要に応じて漢字を併記できる柔軟な制度が整備された。

日本の常用漢字政策が表記の合理化と効率化を目指したのに対し、韓国のハングル専用化は民族的アイデンティティの確立という政治的・社会的な意味合いが強かった。この違いは、両国の言語政策における根本的なアプローチの違いを示している。

現在の韓国の経験は、言語政策が文化や伝統とのバランスを考慮するだけでなく、実際のコミュニケーションにおける機能性や、専門分野での知識伝達の効率性なども含めた総合的な視点が必要であることを示唆している。

高度経済成長期から国際化の時代へ

常用漢字表が制定された1981年という時代は、日本が高度経済成長期を経て、国際社会における存在感を高めていた時期と重なる。この常用漢字表への移行は、1966年から始まった当用漢字表の見直しの結果であった。当用漢字は戦後の漢字制限政策の一環として、漢字の負担を極力減らそうという考えによる制限的な性格が強く、実社会での需要とのズレが生じていた。

新聞社や放送局では適切な漢字使用ができない状況が発生し、法律用語など専門的な分野での漢字使用にも制約があった。このような状況を受けて、1972年から字種の見直しが本格的に検討され始め、1977年に「新漢字表試案」として1,900字が提案され、1979年に「常用漢字表案」として1,926字が中間答申された後、1981年に最終的に1,945字の常用漢字表として制定された。

高度経済成長の終焉と安定成長への移行期において、1970年代のオイルショックを経て、日本経済は安定成長期に入っていた。経済の成熟化に伴い、社会全体の効率化や合理化への関心が高まる中、日本語ワープロの登場と普及により、漢字使用環境も大きく変化していた。1978年に東芝が世界初の日本語ワープロJW-10を発表し、その後各社が追随する形で日本語文書作成の電子化が進んでいった。

また、経済のグローバル化が進み、海外との交流が活発化する中で、国際社会における日本の役割が大きくなるにつれ、日本語表記のあり方についても、国際的な視点が求められるようになった。情報化社会の到来は、文字情報のデジタル化を促進し、日本語の文字コードや入力方式などの標準化も重要な課題となっていた。
このような背景のもと、常用漢字表は当用漢字表と異なり、漢字使用の「目安」として位置づけられ、制限的ではなく必要に応じて表外漢字も使用可能とする、より実用的な指針として確立された。これにより、教育現場や実社会での需要に応じた、より柔軟な漢字政策が展開されることとなったのである。

2010年の常用漢字表改定

常用漢字表は、1981年の制定以来、初めて、2010年（平成22年）に大幅な見直しが行われた。この改定には、以下のような特徴がある。

- **196字の追加と5字の削除**: 情報機器の普及や社会情勢の変化に対応するため、新たに196字が追加され、使用頻度の低い5字が削除された。これにより、常用漢字表の収録字数は2,136字となった。
- **「(参考)」欄の新設**: 従来の「付表」は廃止され、表外漢字・表外音訓は「(参考)」欄に示すこととなった。
- **字体・音訓の整理**: 字体や音訓についても、より実態に即したものに改められた。

これらの変更に加えて、2010年改定の大きな特徴として、**「表外漢字字体表」の考え方を取り入れ、改定常用漢字表と併せて使うことで、印刷標準字体が適用される漢字の範囲が常用漢字以外にも及ぶようにしたこと**が挙げられる。

表外漢字字体表（2000年答申）

2000年（平成12年）12月、国語審議会は「表外漢字字体表」を答申した。これは、常用漢字表に含まれない漢字（表外漢字）について、**印刷文字における字体のよりどころ**を示すことを目的としたものであった。

- **背景**: 常用漢字表の制定以降も、人名や地名、専門用語などには、常用漢字表に含まれない漢字が多く使われていた。これらの漢字の字体には、ばらつきが見られ、情報機器で扱う際にも不便が生じていた。
- **目的**: 表外漢字について、印刷文字における字体選択のよりどころを示すことで、**字体の混乱を抑え、文字情報の円滑な利用に資する**ことを目的とした。

- **3つの字体**: 表外漢字字体表では、表外漢字1,022字について、①いわゆる康熙字典体、②印刷標準字体、③簡易慣用字体の3つの字体が示された。このうち、②印刷標準字体を「印刷文字における現代の通用字体」とし、情報機器の文字集合の漢字の包摂規準として、この印刷標準字体を採用することが望ましいとした。

2010年改定と「表外漢字字体表」との関係

2010年の常用漢字表改定では、この「表外漢字字体表」の考え方を採用し、**「改定常用漢字表」と「表外漢字字体表」を併せて使うことで、印刷標準字体が適用される漢字の範囲が常用漢字以外にも及ぶようにした**。これは、**「表外漢字字体表」が対象としていた表外漢字にも、印刷標準字体に基づく漢字の字体が望ましい字体のよりどころである**としたものである。

つまり、常用漢字については「改定常用漢字表」をよりどころとし、それ以外の漢字については「表外漢字字体表」をよりどころとすることで、**使用頻度の高い漢字の字体について、統一的な指針を示した**のである。

字体の見直し

2010年の改定では、常用漢字表に掲げられた漢字の字体そのものについても、以下のような見直しが行われた。

- **3部首の変更**：「辶（しんにょう）」、「艹（くさかんむり）」、「礻（しめすへん）」の3つの部首の字体が、印刷標準字体に変更された。
- **簡易慣用字体**: 簡易慣用字体とは、表外漢字字体表に示された文字のうち、使用実態を踏まえて印刷標準字体と入れ替えて使用してもよいと判断された文字である。表外漢字字体表（2000年12月の国語審議会答申）では、1,022文字のうち22文字について簡易慣用字体が併記された。2010年の常用漢字表改定では、「曽」「痩」「麺」の3字について、頻度数よりも生活漢字としての側面を重視し、印刷標準字体「曾」「瘦」「麵」ではなく、簡易慣用字体を採用することとなった。

これらの字体の見直しは、印刷文字における字体の統一を図り、情報機器における文字情報の円滑な利用に資することを目的としたものである。

常用漢字の現在と未来

常用漢字表は、現代の日本語表記における重要な指針として、広く用いられている。しかし、情報技術の進展や国際化のさらなる進展など、社会の変化に伴い、今後も見直しが求められる可能性がある。

特に、インターネットや携帯電話の普及により、日常的に接する漢字の範囲が広がっていることは、常用漢字表のあり方に大きな影響を与える可能性がある。また、外国人との交流が活発化する中で、日本語学習者の視点も、今後の漢字政策を考える上で重要となるだろう。

日本における漢字施策と文字集合の歴史

年代	出来事	文字数
1904年	合計857字の**教授漢字**を制定	857字
1923年	文部省臨時国語調査会 **常用漢字表** 発表	1,962字
1940年	日本陸軍 **兵器名称用制限漢字表** 制定	1,235字
1942年	国語審議会 **標準漢字表** 答申	2,528字
1946年	**当用漢字表** 内閣告示	1,850字
1947年	**日本国憲法・法令における当用漢字の使用開始**	
1948年	**人名用漢字** 制定	92字
1949年	**当用漢字音訓表** 告示	
1973年	**当用漢字改定案**	
1981年	内閣告示第1号 **常用漢字表** 公布	1,945字
2004年	**人名用漢字 大幅改訂**	488字
2000年	**表外漢字字体表** 答申	1,022字
2010年	**常用漢字表改定**	2,136字

文字コードの歴史

コンピュータは、内部ではすべての情報を数値（0と1の組み合わせ）で処理している。そのため、人間が使う文字も、コンピュータで扱うためには、数値に変換（符号化）する必要がある。**文字コード**とは、文字や記号などの各キャラクタに対して、固有の数値（符号）を割り当てるルール、またはそのルールに基づいて文字を数値に変換した結果（符号）を指す。文字をコード化する主な目的は、以下のとおりである。

- **コンピュータでの処理を可能にする**: コンピュータは、すべての情報を数値として扱う。文字をコード化することで、コンピュータで文字情報を処理、保存、伝送することが可能になる。
- **情報の共有を可能にする**: 異なるコンピュータやシステム間でも、共通の文字コードを用いることで、文字情報を正確にやり取りすることができる。
- **多言語対応を可能にする**: 世界中の文字を統一的に扱う文字コード（Unicodeなど）を用いることで、多言語環境での情報処理が容易になる。

初期の文字コード

日本語をコンピュータで扱うための試みは、1960年代から始まった。当初は、電算写植システムごとに独自の文字コードが用いられていたが、やがて標準化の必要性が認識され、日本工業標準調査会（JISC）によって、統一的な文字コード規格が制定されるようになった。

コンピュータによる情報処理以前の電気通信の分野では、アルファベットを表現するモールス符号や、仮名と若干の漢字を表現するテレックス用文字コードが存在した。モールス符号は黒船来航とともに日本にもたらされ、明治初期には日本国内でも電信網が構築された。

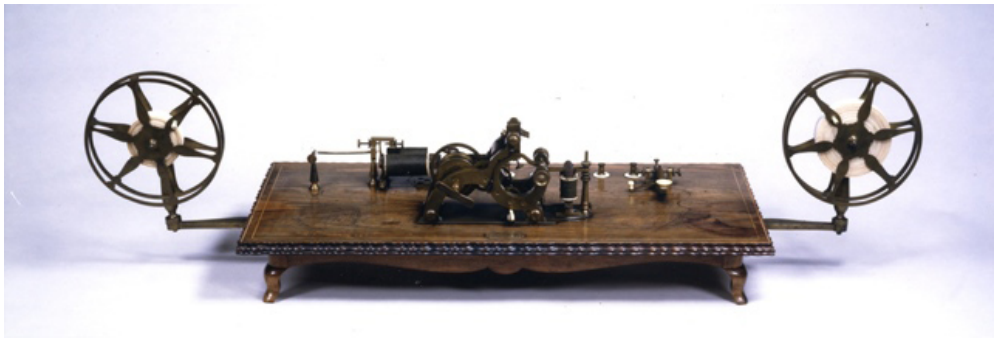


図1: ペリー来航時 米大統領が幕府に献上したエンボッシング・モールス電信機（郵政博物館）

英文モールス符号

文字	符号	文字	符号	文字	符号	文字	符号
A	・－	B	－・・・	C	－・－・	D	－・・
E	・	F	・・－・	G	－－・	H	・・・・
I	・・	J	・－	K	－・－	L	・－・・
M	－－	N	－・	O	－－－	P	・－－・
Q	－－・－	R	・－・	S	・・・	T	－
U	・・－	V	・・・－	W	・－－	X	－・・－
Y	－・－	Z	－－・・	0	－－－－－	1	・－
2	・・－－－	3	・・・－－	4	・・・・－	5	・・・・・
6	－・・・・	7	－－・・・	8	－－－・・	9	－－－－・
.	・－・－・－	,	－－・・－－	?	・・－－・・	'	・－・－・
!	－・－・－－	/	－・・－・	(	－・－・	)	－・－・－
&	・－・・・	:	－－－・・・	;	－・－・－・	=	－・・・－
+	・－・－・	-	－・・・・－	_	・・－－・－	"	・－・・－・
$	・・・－・・－	@	・－－・－・				

漢字を電気通信で送る初期の試みとしては、漢字を4桁または5桁の数字で表現し、それをモールス符号で送信する方法が用いられた。また、漢字テレタイプなどの機器も開発された。これらの技術は、日本語の電気通信における漢字表現の先駆けとなったが、表現できる漢字の数や通信速度に限界があった。

情報処理の始まりと文字コードの必要性

コンピュータによる情報処理の黎明期、日本語をどのように表現するかは大きな課題であった。当時は、コンピュータの性能が低く、メモリ容量も限られていたため、扱うことができる文字数には大きな制約があった。また、漢字、ひらがな、カタカナという複数の文字種を持つ日本語を、どのようにコード化するかという問題もあった。

この頃、海外では、米国を中心に文字コードの標準化が進められていた。1963年には、米国規格協会（ASA）が ASA X3.4、いわゆる ASCII（American Standard Code for Information Interchange）を制定した。ASCIIは、7ビットで英数字、記号など128文字を表現する文字コードであり、その後の情報処理における基本的な文字コードとなった。

American Standard Code for

Information Interchange

1. Scope

This coded character set is to be used for the general interchange of information among information processing systems, communication systems, and associated equipment.

2. Standard Code

b_7				0	0	0	0	1	1	1	1
b_6				0	0	1	1	0	0	1	1
b_5				0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1								
0	0	0	0	NULL	DC_0	b	0	@	P		
0	0	0	1	SOM	DC_1	!	1	A	Q		
0	0	1	0	EOA	DC_2	"	2	B	R		
0	0	1	1	EOM	DC_3	#	3	C	S		
0	1	0	0	EOT	DC_4 (STOP)	$	4	D	T		
0	1	0	1	WRU	ERR	%	5	E	U		
0	1	1	0	RU	SYNC	&	6	F	V		
0	1	1	1	BELL	LEM	' (APOS)	7	G	W		
1	0	0	0	FE_0	S_0	(	8	H	X		
1	0	0	1	HT / SK	S_1	)	9	I	Y		
1	0	1	0	LF	S_2	*	:	J	Z		
1	0	1	1	V_{TAB}	S_3	+	;	K	[		
1	1	0	0	FF	S_4	, (COMMA)	<	L	\		ACK
1	1	0	1	CR	S_5	−	=	M	]		①
1	1	1	0	SO	S_6	.	>	N	↑		ESC
1	1	1	1	SI	S_7	/	?	O	←		DEL

UNASSIGNED

UNASSIGNED

図2: ASA X-3.4-1963

ASCIIでは、制御文字、スペース、数字、アルファベットの大文字・小文字、およびいくつかの記号が定義されている。

制御文字とは

ASCIIコード表やISO/IEC 646コード表、そしてそれらを基にしたJIS X 0201のコード表には、画面に表示される文字（印字文字）だけでなく、**制御文字**と呼ばれる特殊な役割を持つ文字が含まれている。

制御文字は、その名の通り、通信や機器、ソフトウェアの動作を「制御」するために用いられる。現代のパソコンでは、画面に表示されることもなく、その機能が意識されることも少なくなった。しかし、当時はテレタイプ端末や、シリアルポートで接続されたタイプライター状のコンソールなどを用いてシステムとやり取りをしていた。

テレタイプ端末とは

テレタイプ端末は、タイプライターと通信機能を組み合わせたような入出力装置であり、キーボードから入力した文字を送信したり、受信した文字を紙に印字したりすることができた。また、テレタイプ端末自体がコンピュータの入出力装置として用いられることもあった。

図3: 日本勧業銀行で本・支店間の通信に使用されたテレタイプ（1967年）

テレタイプ端末では、制御文字は、以下のような動作を引き起こす。

- **BEL (Bell)**: 端末のベル（実際には電子ブザー音）を鳴らすことで、オペレータに注意を促す。
- **BS (Backspace)**: 紙に印字するヘッドや打鍵位置を1文字分戻す。
- **HT (Horizontal Tab)**: あらかじめ設定された位置（タブ位置）まで印字ヘッドを移動させる。
- **LF (Line Feed)**: 紙を一行分送る。
- **VT (Vertical Tab)**: あらかじめ設定された行（垂直タブ位置）まで紙を送る。
- **FF (Form Feed)**: 次のページの先頭まで紙を送る。
- **CR (Carriage Return)**: 印字ヘッドを行の先頭（キャリッジ）に戻す。

現代では制御文字が直接機器を制御することはほとんどない。しかし、ファイルやデータの中に制御文字が含まれていることがあり、特に改行を示すLFやCRは、システム間でテキストデータをやり取りする際に意識する必要がある。

例えば、以下のようなものがある。

- **NUL (Null)**: 何もしないことを示す。初期のシステムでは時間稼ぎやデータの埋め草として使われた。
- **SOH (Start of Heading)**: ヘッダー（データの先頭にある付加情報）の開始を示す。
- **STX (Start of Text)**: テキスト（本文）の開始を示す。
- **ETX (End of Text)**: テキストの終了を示す。
- **EOT (End of Transmission)**: 通信の終了を示す。
- **DEL (Delete)**: 文字の削除

これらの制御文字は、現代ではあまり使われなくなったものもある。しかし、例えばLF（改行）やCR（復帰）は、改行コードとして現在でも広く使われている。また、通信プロトコルやファイルフォーマットの中で、特定の意味を持つ制御文字が使われることもある。

注記:

- Windowsでは、改行コードとしてCRとLFの組み合わせ（CR+LF）が一般的に使われる。
- UNIX系OS（LinuxやmacOSなど）では、改行コードとしてLFのみが使われる。
- このように、OSによって改行コードが異なるため、テキストファイルを異なるOS間でやり取りする際に、改行が正しく表示されないことがある。

ASCIIを基にした国際規格ISO/IEC 646の制定

さらに、1967年には、国際標準化機構（ISO）と国際電気標準会議（IEC）が、ASCIIを基にした国際規格ISO/IEC 646を制定した。ISO/IEC 646は、7ビットの文字コードであり、基本的な枠組みはASCIIと同じだが、一部の記号を各国で異なる文字を割り当てられるように、国別文字用のコードポイントとして予約していた。

ISO 646-1973 (E)

3 BASIC CODE TABLE

TABLE 1 — Basic code table

b_4	b_3	b_2	b_1	row \ column	0	1	2	3	4	5	6	7
				b_7	0	0	0	0	1	1	1	1
				b_6	0	0	1	1	0	0	1	1
				b_5	0	1	0	1	0	1	0	1
0	0	0	0	0	NUL	TC_1 (DLE)	SP	0	@	P	` [4]	p
0	0	0	1	1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0	0	1	0	2	TC_2 (STX)	DC_2	" [4]	2	B	R	b	r
0	0	1	1	3	TC_3 (ETX)	DC_3	£(#) [2]	3	C	S	c	s
0	1	0	0	4	TC_4 (EOT)	DC_4	$(¤) [2]	4	D	T	d	t
0	1	0	1	5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0	1	1	0	6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0	1	1	1	7	BEL	TC_{10} (ETB)	' [6]	7	G	W	g	w
1	0	0	0	8	FE_0 (BS)	CAN	(	8	H	X	h	x
1	0	0	1	9	FE_1 (HT)	EM	)	9	I	Y	i	y
1	0	1	0	10	FE_2 (LF)[0]	SUB	*	:	J	Z	j	z
1	0	1	1	11	FE_3 (VT)[0]	ESC	+	;	K	[3]	k	[3]
1	1	0	0	12	FE_4 (FF)[0]	IS_4 (FS)	, [6]	<	L	[3]	l	[3]
1	1	0	1	13	FE_5 (CR)[0]	IS_3 (GS)	-	=	M	[3]	m	[3]
1	1	1	0	14	SO	IS_2 (RS)	.	>	N	^ [3][5]	n	~ [3][5]
1	1	1	1	15	SI	IS_1 (US)	/	?	O	_	o	DEL

2

図4: ISO646-1973 3. TABLE 1 - BASIC CODE TABLE

各国別文字の例：

符号位置	ISO/IEC 646	独 DIN 66003	仏 NF Z 62-010	英 BS 4730	日 JIS C 6220-1969
0x23	#	§	§	£	#
0x24	$	¤	$	$	¥
0x40	@	@	à	@	@
0x5B	[	Ä	é	[	「
0x5C	\	Ö	ù	\	\
0x5D	]	Ü	è	]	」
0x5E	^	^	^	^	¯
0x60	`	`	`	`	`
0x7B	{	ä	ç	#	{
0x7C	|	ö	°	|	|
0x7D	}	ü	à	£	}
0x7E	~	~	²	~	-

注記：

- 日本の欄に記載されている文字は、JIS C 6220-1969のラテン文字集合で規定されたものである。
- 0x5Cは、後のJIS X 0201では'¥'（円記号）となる。ただし、ASCIIやISO/IEC 646国際基準版との互換性維持のため、Unicodeでは'\'（バックスラッシュ）と同じU+005Cに統合されている。
- 0x7Eは、後のJIS X 0201では'‾'（オーバーライン）となる。ただし、ASCIIやISO/IEC 646国際基準版との互換性維持のため、Unicodeでは'~'（チルダ）と同じU+007Eに統合されている。

このように、ISO/IEC 646では、一部のコードポイントを各国で自由に割り当てられるようにすることで、ウムラウト（例：Ä, Ö, Ü）やアクサン記号付き文字（例：é, à, è）など、ASCIIでは表現できなかった文字を表現することが可能となった。しかし、日本語のひらがなやカタカナ、漢字を表現するには、7ビットでは全く不十分であった。

JEIDAによる仮名文字の標準符号制定

日本でも、このような国際的な標準化の動きを受け、JISCを中心に標準化の検討が始まった。初期の日本語処理システムでは、各メーカーが独自の文字コードを開発・採用していた。しかし、これらの文字コードは互換性がなく、異なるシステム間でのデータ交換は困難であった。

1969年、日本電子工業振興協会（JEIDA、現在の電子情報技術産業協会）が、カナ文字の標準符号を制定した。これは、ASCIIの空き領域、具体的には8ビット目を立てたエリアにカタカナなどを配置した8ビットの文字コードであった。この時点では漢字は含まれていない。

この後、JIS C 6220（後のJIS X 0201）が制定され、ラテン文字と片仮名の集合が規定された。これらの初期の試みは、日本語情報処理の基礎を築くと同時に、漢字を含むより大規模な文字集合の必要性を浮き彫りにした。

全銀テレ為替文字

初期の文字コードが、現代でも使われ続けている身近な例として、**全国銀行データ通信システム**（全銀システム）で用いられている**全銀テレ為替文字**が挙げられる。全銀システムは、銀行間の内国為替取引をオンラインで中継するシステムであり、1973年に運用が開始された。このシステムで用いられる文字コードは、**全銀協標準通信プロトコル（ベーシック手順）**と共に、現在でも多くの場面で利用され続けている。

全銀テレ為替文字は、JIS X 0201と類似した構成を持つ、8ビットの文字コード体系である。具体的には、ASCIIをベースに、英小文字・一部の記号を、カタカナ・濁点・半濁点に置き換えたものである。漢字は含まれない。

特徴:

- 英大文字、数字、カタカナ（濁点、半濁点を含む）、一部の記号（`@`, `*`, `¥` など）を表現できる。
- 濁点、半濁点は独立したコードポイントを持つ。
- ひらがな、漢字は含まれない。

現代における利用例:

- 銀行間のデータ交換
- 企業と銀行間のデータ交換（ファームバンキング）
- レガシーシステムとのデータ連携

全銀テレ為替文字は、現代のUnicode全盛の時代においても、金融システムを支える重要な役割を果たし続けている。これは、初期の文字コードが、単に過去の遺物ではなく、現代社会の基盤の一部として生き続けていることを示す好例といえる。ただし、将来的には、より多くの文字を表現できるUnicodeなどへの移行が進められていくことが予想される。

ISO-2022

7ビットの文字コード環境において、日本語のような多種多様な文字を扱うために開発されたのがISO-2022という規格である。ISO-2022は、複数の文字集合を切り替えて使用するための国際規格であり、エスケープシーケンスと呼ばれる特殊なコード列を使って、文字集合を切り替える仕組みを持つ。これにより、7ビット環境でも、JIS X 0201、JIS X 0208、JIS X 0212など、様々な文字集合を組み合わせて使用することが可能となった。

ISO-2022は、主に電子メールなどのテキストデータで利用され、特に日本語環境ではISO-2022-JPというプロファイルが広く使用された。ISO-2022-JPは、ASCIIコードとJIS X 0208の文字を組み合わせて利用できるように定義されたものである。

しかし、ISO-2022はエスケープシーケンスが多いため、テキストの可読性が低く、また、処理も複雑になるという課題があった。Unicodeの普及に伴い、ISO-2022は徐々に使われなくなり、現在ではレガシーシステムや一部のメールシステムを除いて、ほとんど利用されていない。

汎用機時代の日本語文字コード

1970年代から1980年代にかけて、企業の情報処理の中心は汎用機（メインフレーム）が担っていた。各コンピュータメーカーは、自社の汎用機で日本語を扱うために、独自の文字コードを開発した。これらの文字コードは、JIS C 6226をベースに、各社が独自の拡張を施したものであった。

汎用機（メインフレーム）とは、大規模なデータ処理を目的として設計された大型コンピュータである。汎用機は、高い信頼性、処理能力、入出力性能を備え、企業の基幹業務や科学技術計算などに用いられてきた。汎用機は、専用のオペレーティングシステム（OS）で動作し、多数の端末から同時にアクセスできるマルチユーザー・マルチタスク環境を提供する。また、バッチ処理やオンライン処理など、さまざまな形態のデータ処理に対応している。

図5: 労働省職業安定局労働市場センターにてFACOM 230（1965年）

JIS C 6226（旧JIS漢字）

日本で最初の統一的な日本語文字コード規格は、1969年に制定された**JIS C 6226-1969**（情報交換用漢字符号）である。この規格では、漢字を含む約3,000文字が収録され、当時の当用漢字がほぼ網羅されていた。

JIS C 6226は、当初、電算写植システムでの利用を想定して制定された。電算写植システムとは、コンピュータ制御によって自動的に活字を拾い、組版を行うシステムである。JIS C 6226は、電算写植システムにおける日本語の表現方法を標準化する役割を果たした。

その後、1974年には、JIS C 6226の適用範囲を印刷・情報処理用まで広げ、水準の概念を導入した**JIS C 6226-1974**（情報交換用漢字符号系）が制定された。

1978年には、JIS C 6226が全面的に改訂され、**JIS C 6226-1978**（情報交換用漢字符号系）が制定された。この規格では、第1水準漢字（2,965字）と第2水準漢字（3,384字）、そして非漢字（記号、ひらがな、カタカナなど）を含む、合計6,802文字が収録された。JIS C 6226-1978は、その後の日本語文字コードの基礎となり、長く使われることとなる。一般に**旧JIS漢字**と呼ばれている。

汎用機における日本語文字セット

主要な汎用機メーカーおよび日本電信電話公社は、それぞれ独自の日本語文字セットを開発した。代表的なものとして、**IBMのIBM漢字コード**、**日立のKEISコード**、**富士通のJEFコード**、**NECのJIPSコード**、**日本電信電話公社のDIPS漢字コード**が挙げられる。これらの文字コードは、それぞれのメーカーのメインフレーム（汎用機）で使用され、当時の日本の情報処理基盤を支えていた。

EBCDIC (Extended Binary Coded Decimal Interchange Code)

EBCDIC（イビシディック）は、IBMが開発した文字コード体系である。主にIBMの汎用機で使用されている。EBCDICは、8ビットの符号で文字を表現し、ASCIIとは異なる体系を持っている。

日本語を表現するために、EBCDICを拡張したコード体系（EBCDIK：カタカナ・コードなど）も開発された。しかし、EBCDICはASCIIとの互換性がないため、汎用機とパソコンなどの間でデータをやり取りする際には、コード変換が必要となる。

IBM: IBM漢字コード

IBM漢字コードは、IBMが開発した日本語文字セットおよびその符号化方式である。**IBM日本語カタカナ・漢字コード**とも呼ばれ、1971年に発表された。この文字コードは、System/360、System/370などのIBMのメインフレームで使用された。

- **概要と特徴、収録文字**: IBM漢字コードは、**EBCDIC**（Extended Binary Coded Decimal Interchange Code）をベースとしており、カタカナ、英数字、記号、そして約1万2千字の漢字を収録していた。漢字は、JIS X 0208を参考に選定され、独自のコードで収録されていた。また、1983年には、JIS X 0208-1983に対応し、非漢字と漢字を統合したコード体系である**IBM漢字コード(新拡張)**を発表した。
- **ホスト漢字コード**: 1986年には、**ホスト漢字コード**が制定された。これは、従来ASCIIコード空間に独自定義していたカタカナ、英数字と、JIS X 0208で規定された非漢字、漢字を統合したものである。また、1988年には、**パーソナルコンピュータ拡張漢字**、**パーソナルコンピュータ拡張非漢字**が追加され、約1,900字が追加された。

日立: KEIS コード

KEIS（**K**ANJI **E**XTENDED **I**NFORMATION PROCESSING **S**YSTEM、ケイス）**コード**は、日立製作所が開発した日本語文字セットおよびその符号化方式である。KEIS コードは、日立のメインフレーム HITAC で使用された。

- **概要と特徴、収録文字**: KEIS コードは、8 ビットおよび 16 ビットコード体系であり、JIS X 0208 を基に拡張され、約 8,000 字の漢字を収録していた。漢字の配列は、JIS X 0208 とは異なり、独自の配列を採用していた。JIS X 0208 で規定された非漢字、漢字に加え、日立独自の拡張漢字約 1,400 字、利用者定義文字領域約 2,000 字が追加されている。
- **EBCDIK との関係**: KEIS コードは、日立製作所による EBCDIC コードの拡張である **EBCDIK** コードを使用し、カタカナ、英数字領域の制御コードの一部が、EBCDIK におけるコードと同一である。

富士通: JEF コード

JEF（**J**APANESE PROCESSING **E**XTENDED **F**EATURE、ジェフ）**コード**は、富士通が開発した日本語文字セットおよびその符号化方式である。JEF コードは、富士通のメインフレーム FACOM で使用された。

- **概要と特徴、収録文字**: JEF コードは、富士通のメインフレームである FACOM M シリーズで使用されていた、16 ビットコード体系の日本語文字コードである。JIS X 0208 の漢字集合を基に、約 1 万 2 千字の漢字を収録し、その他、利用者定義文字や富士通独自の拡張文字、制御文字なども定義していた。1979 年に発表された当初の規格では、約 6,800 字の漢字を収録していたが、1985 年に発表された拡張 JEF では、JIS X 0208-1983 に対応し、さらに拡張漢字を追加し、約 1 万 2 千字の漢字を収録するようになった。

日本電気: JIPS (Japanese Information Processing System)

JIPS（ジップス）は、日本電気が開発した日本語文字セットおよびその符号化方式である。JIPS は、NEC のメインフレーム ACOS シリーズで使用された。

- **JIPS の概要と特徴、収録文字**: JIPS は、JIS C 6226(JIS X 0208 の旧規格)を基に拡張され、約 7,000 字の漢字を収録していた。JIS C 6226 に含まれる漢字のほか、NEC 独自の拡張文字も含まれていた。また、漢字の配列は JIS C 6226 とは異なり、漢字の構成要素（部首など）に基づいた独自の配列が採用されていた。
- **JIPS(E), JIPS(J), JIPS(A)**: JIPS には、JIS 配列による **JIPS(J)**、構成要素順配列の **JIPS(E)**、ASCII 配列の **JIPS(A)** といった種類があった。

日本電信電話公社: DIPS漢字コード

DIPS漢字コードは、日本電信電話公社（現NTT）が開発したメインフレームであるDIPS上で使用されていた日本語文字コードである。

- **DIPS漢字コードの概要と特徴、収録文字**: DIPS漢字コードは、JIS C 6226を基に拡張され、漢字、仮名、英数字などを収録していた。漢字の配列は、JIS C 6226とは異なり、独自の配列が採用されていた。また、DIPS漢字コードには、日本電信電話公社が独自に追加した外字も含まれていた。

これらの汎用機メーカー独自の日本語文字セットは、それぞれの企業の顧客を中心に広く利用され、日本の情報処理の発展に大きく貢献した。しかし、異なるメーカー間でのデータ互換性に課題があり、後のJIS X 0208の普及やUnicodeへの移行の背景となった。

汎用機における日本語エンコーディング

汎用機では、日本語文字を表現するために、各社が独自の符号化方式を定めている。これらの符号化方式では、漢字や仮名などの2バイト文字と、ASCIIなどの1バイト文字を混在して扱うために、シフトコードやエスケープシーケンスなどの特殊な制御コードが用いられる。

シフトコード、エスケープシーケンスを用いた表現

- **シフトコード**: 1バイト文字と2バイト文字の切り替えを指示する特殊なコードである。例えば、Shift_JISでは、0x80以上のコードを漢字の1バイト目として使用し、それ以降の2バイトで1文字を表現する。

- **エスケープシーケンス**: 文字コードの体系を切り替えるための特殊なコード列である。ISO-2022では、エスケープシーケンスを用いて、ASCII、JIS X 0208、JIS X 0212などを切り替えて使用する。ISO-2022では、以下のようなエスケープシーケンスが定義されている。

  - `ESC $ B`: JIS X 0208:1983 (JIS83) に切り替える
  - `ESC ( B`: ASCIIに切り替える
  - `ESC $ @`: JIS X 0208:1978 (JIS78) に切り替える
  - `ESC ( J`: JIS X 0201のローマ字を呼び出す
  - `ESC ( I`: JIS X 0201の片仮名を呼び出す

これらのエスケープシーケンスは、テキストデータ中に現れ、その後に続く文字をどの文字集合で解釈すべきかを指定する。これにより、ISO-2022は、複数の文字集合を混在させた文書を表現することを可能にした。

文字集合、符号化方式の差異と文字化け

汎用機メーカーごとに、文字セットや符号化方式が異なるため、異なるメーカーの汎用機間でデータを交換する際には、文字コードの変換が必要となる。しかし、各社の文字コードには、収録文字の範囲や符号位置、符号化方式に差異があるため、完全な変換ができない場合がある。特にJIS規格から各ベンダーが独自に拡張した文字や、顧客組織が個別に追加した「外字」の扱いは、各社で異なるため、問題となりやすい。

文字コード変換を行う際には、以下のような問題が発生する可能性がある。

- 一方の文字コードに含まれている文字が、他方の文字コードに含まれていない場合、その文字を表現することができない。
- 一方の文字コードで表現されていた文字が、他方の文字コードでは異なる字形に対応付けられている場合、文字の見た目が変わってしまう。
- 一方の文字コードで用いられていた外字が、他方の文字コードでは定義されていない場合、その外字を表現することができない。

文字コードの変換に失敗すると、いわゆる「文字化け」が発生する。文字化けとは、元の文字とは異なる文字が表示されたり、文字が全く表示されなくなったりする現象である。文字化けが発生すると、データの正確性が損なわれ、業務に支障をきたす可能性がある。

オープンシステムへの移行

1990年代以降、汎用機からオープンシステムへの移行が進んだ。オープンシステムとは、異なるベンダーの製品や技術を組み合わせて構築された情報システムのことである。オープンシステムでは、業界標準の技術やプロトコルが用いられるため、ベンダー依存性を排除することができる。

オープンシステムへの移行に伴い、汎用機で用いられていたベンダー独自の文字コードは、徐々に使われなくなっていった。しかし、現在でも、一部の企業では汎用機が稼働しており、過去のデータ資産を継承するために、汎用機コード体系が用いられている場合がある。

パソコン時代の文字コード

1980年代以降、パソコンやUNIXワークステーションが普及し、企業の情報処理環境は大きく変化した。パソコンやUNIXでは、汎用機とは異なる文字コードが用いられるようになった。

パソコンの普及と日本語処理の進化

1980年代初頭、NECのPC-8801やPC-9801シリーズなどの国産パソコンが登場し、日本独自のパソコン市場が形成された。これらのパソコンでは、当初は限られた文字数の日本語表示しかできなかったが、やがて漢字ROMを搭載し、JIS漢字を扱えるようになった。

パソコンの普及に伴い、日本語ワープロ専用機も登場した。ワープロ専用機は、文書作成に特化したコンピュータであり、日本語入力や編集のための機能を備えていた。ワープロ専用機は、パソコンよりも安価で操作が容易であったため、企業や個人に広く普及した。

JIS X 0208（新JIS漢字）

JIS X 0208は、1983年（昭和58年）に制定された、7ビット及び8ビットの2バイト情報交換用符号化漢字集合の規格である。一般に**新JIS漢字**と呼ばれる。

- **背景と歴史**: JIS X 0208は、JIS C 6226を改訂する形で制定された。JIS C 6226の符号化方式を拡張し、より多くの漢字を収録できるようにしたものである。
- **制定年**: 1983年（昭和58年）
- **利用分野**: 情報処理システム、特にパソコンやUNIXシステムにおける日本語処理
- **文字数**: 6,879文字（漢字6,355字、非漢字524字）
- **特徴**: JIS X 0208は、JIS C 6226と同様に、漢字を第1水準と第2水準に分けて収録している。また、JIS C 6226に含まれていた非漢字に加えて、罫線素片などの記号が追加されている。JIS X 0208は、**EUC-JP**や**Shift_JIS**などの符号化方式で用いられている。
- **変遷**: JIS X 0208は、1990年（平成2年）と1997年（平成9年）に改訂されている。1990年の改訂では、いくつかの記号が追加された。1997年の改訂では、字体の変更や、いわゆる機種依存文字への対応などが行われた。

Shift_JIS: パソコン環境での日本語処理

Shift_JIS（シフトジス）は、マイクロソフトとアスキー（現：株式会社アスキー）が開発した日本語文字コード（符号化方式）である。パソコン用の文字コードとして、広く普及した。

開発の背景と特徴

Shift_JISは、JIS X 0208の漢字を、既存のASCIIコード（1バイト文字）と共存できるように符号化したものである。Shift_JISでは、JIS X 0208の漢字を表現するために、2バイトを使用する。ただし、1バイト目には、ASCIIでは使用されていない領域（0x81～0x9F、0xE0～0xFC）を割り当てることで、ASCIIコードとの判別を可能にしている。

常用漢字との関係

Shift_JISは、JIS X 0208を符号化しているため、JIS X 0208に含まれる常用漢字を表現することができる。ただし、JIS X 0208制定後に常用漢字に追加された漢字は、Shift_JISの標準の範囲には含まれていない。

外字問題と独自拡張

Shift_JISでは、JIS X 0208に含まれない文字（外字）を表現するために、ユーザーが独自に文字を定義できる領域（外字領域）が用意されている。しかし、外字領域の利用方法は、ベンダーやシステムによって異なるため、互換性の問題が発生しやすい。

また、一部のベンダーは、Shift_JISに独自の拡張を施している。例えば、NEC特殊文字、IBM拡張漢字などがある。これらの拡張文字は、異なるベンダーのシステム間では、正しく表示されない場合がある。

EUC-JP: UNIX環境での日本語処理

EUC-JP（Extended Unix Code Packed Format for Japanese）は、UNIXで日本語を扱うために開発された文字コード（符号化方式）である。EUC-JPは、JIS X 0208の漢字を2バイトで表現し、JIS X 0212の補助漢字を3バイトで表現する。また、ASCII文字を1バイトで表現できるため、ASCIIとの親和性が高い。

EUC-JPは、主にUNIXワークステーションやサーバーで用いられてきた。しかし、近年では、Unicode（特にUTF-8）の普及により、EUC-JPの使用頻度は減少している。

JIS X 0212（補助漢字）

JIS X 0212は、1990年（平成2年）に制定された、情報交換用漢字符号ー補助漢字の規格である。一般に**補助漢字**と呼ばれる。

- 1978年に制定されたJIS C 6226（後のJIS X 0208）は、当時の日本語情報処理で必要とされる漢字を6,349字（第1水準2,965字、第2水準3,384字）収録していた。しかし、制定後、情報化の進展と共に、JIS X 0208に含まれない漢字（人名用漢字、地名、古典、方言、学術用語などに用いられる漢字）へのニーズが高まっていった。

- 1980年代には既に、各省庁や自治体等がそれぞれに外字を定義して使用している状況が生じており、行政事務で用いられる人名・地名用漢字、学術研究で利用される古典籍などに登場する異体字への早急な対応が求められていた。

- これを受けて、1983年にJIS X 0208が改定されたものの、この改定で追加・変更された文字はわずかであり、上記のニーズを満たすには至らなかった。

- このような背景のもと、JIS X 0208を補完する目的で、1990年にJIS X 0212（補助漢字）が制定された。

- **制定年**: 1990年（平成2年）

- **利用分野**: JIS X 0208に含まれない漢字を必要とする情報処理システム（例：行政事務、学術研究、出版など）

- **文字数**: 6,067文字（漢字5,801字、非漢字266字）

補助漢字に追加された文字の例

- **人名用漢字・地名**:「吉（「吉」の異体字）」「峪（「栃」の異体字）」「鮍（「鱈」の異体字）」「桒」「髙」「濵」「原」「塚」など
- **学術用語**:「鎩」「鑛」「鉱」「鮓」「麭」など
- **古典・方言**:「儁」「石」「烋」「槝」「洩」など
- **その他**:「烽」「鄙」「朖」など

補助漢字の特徴

- JIS X 0212は、JIS X 0208とは独立した文字集合であり、JIS X 0208と組み合わせて使用されることを想定している。
- 補助漢字は、EUC-JPでは3バイトで符号化される。
- 補助漢字の選定にあたっては、上記のような人名、地名、学術用語などで使用実績のある漢字が調査され、JIS X 0208との重複を避けて収録された。

補助漢字制定後の状況

- JIS X 0212の制定は、**特に人名・地名を扱う行政事務の電子化において大きな意味を持った**。1990年代後半以降、**公的個人認証サービス**をはじめとする多くの電子政府システムで、補助漢字を前提としたシステム構築が行われてきたという経緯がある。
- 近年では、Unicodeの普及により、文字集合を個別に意識する場面は減少しているものの、補助漢字は、JIS X 0208だけではカバーできない漢字を補完する文字集合として、日本の文字コードの歴史において重要な役割を果たしたと言える。また、過去に構築されたシステムとの互換性を維持する上でも、引き続き重要な役割を担っている。

インターネット黎明期の文字コード

インターネット黎明期、日本語のウェブページや電子メールでは、使用される文字コードに複数の選択肢が存在し、環境によって使い分けられていた。ウェブページでは Shift_JIS が広く用いられていたが、学術機関やUNIXワークステーションのユーザーを中心に、EUC-JP が使用されることも多かった。これは、UNIX系OSで標準的に採用されていた文字コードがEUCであったためである。

一方、電子メールでは、7bit のみを使用する JIS コード (ISO-2022-JP) が標準的に利用されていた。これは、当時のインターネットで標準的に使用されていたプロトコルである SMTP (Simple Mail Transfer Protocol) が、7bit のみの文字コードしかサポートしていなかったためである。8bit 以上のコードを送信すると、データが破損する可能性があったため、7bit の範囲内で日本語を表現できる JIS コードが用いられた。ISO-2022-JP は、エスケープシーケンスを用いて文字集合を切り替えることで、ASCII と JIS X 0208 を混在させることができた。

ISO-2022は、7ビット環境で多言語を扱うための重要な技術であったが、エスケープシーケンスが多いため、テキストの可読性が低い、処理が複雑になるなどの問題があり、Unicodeの普及とともに使われなくなっていった。しかし、レガシーシステムとの連携や、電子メールの歴史を理解する上では、ISO-2022の知識は不可欠である。

その他の文字コード規格

これまで説明してきた文字コード以外にも、日本語を表現するために、さまざまな文字コードやデータベースが開発されてきた。例えば、**今昔文字鏡**は、今昔文字鏡研究所が開発した文字データベースおよびその関連ソフトウェアで、漢字を中心に約17万字もの多様な文字を収録している。古代文字や異体字、古典籍で使用された文字など、既存の文字コード規格では表現できない漢字を多数収録している点が特徴で、文字学や歴史学などの研究分野で利用されている。各文字には固有の文字鏡番号が割り当てられ、正確な指定が可能だ。

一方、**TRONコード**は、TRONプロジェクトで開発された文字コード体系で、多言語対応と将来的な文字数の増加への対応を重視し、約18万字の文字を収録している。TRONコードは、BTRON仕様のOSである「超漢字」などで用いられている。BTRONは、パソコンやワークステーション向けのOS仕様であり、多言語対応の先駆けとなった。

図6: BTRON仕様OS「B-right/V®」を含んだデスクトップ環境 超漢字Vの画面

これらの他にも、学術的なプロジェクトとして、京都大学人文科学研究所の**漢字データベース**や、東京大学大学院人文社会系研究科の**大蔵経テキストデータベース(SAT)**など、特定の分野に特化した文字データベースも構築されている。これらは、標準的な文字コードでは網羅されていない特殊な文字や、研究上必要な詳細な文字情報を管理・提供することで、学術研究の発展に寄与している。

現代の文字コード

JIS X 0213（JIS2000, JIS2004）

JIS X 0213は、2000年に制定された日本語文字コードの規格であり、**JIS2000**とも呼ばれる。その後、2004年に改定され、**JIS2004**となった。JIS X 0213は、JIS X 0208（以下、0208と略す）とJIS X 0212（以下、0212と略す）を拡張し、より多くの漢字を収録したものである。

JIS X 0213は、多漢字化への対応と、機種依存文字問題の解決を目的として制定された。0208制定当時に比べて、情報機器で扱える文字数が大幅に増えたこと、また、人名や地名などで、より多くの漢字を表現する必要性が生じたことが背景にある。

- **背景と歴史**: JIS X 0213は、0208と0212に含まれない漢字を収録し、多漢字化への対応を図る目的で制定された。また、0208の改訂で問題となった機種依存文字問題への対応も考慮されている。
- **制定年**: 2000年（平成12年）
- **利用分野**: 多漢字を必要とする情報処理システム
- **文字数**: 11,233文字（漢字10,050字、非漢字1,183字）
- **特徴**: JIS X 0213では、漢字を**第1面**と**第2面**に分けて収録している。第1面には、0208の第1水準と第2水準の漢字に加えて、新たに**第3水準漢字**が追加されている。第2面には、**第4水準漢字**が収録されている。また、非漢字も大幅に拡充されている。JIS X 0213は、**JIS2000**や**JIS2004**とも呼ばれる。
- **変遷**: JIS X 0213は、2004年（平成16年）に改訂され（JIS2004）、10文字の字形が変更され、163文字の例示字形が変更された。

収録漢字と利用用途

JIS X 0213は、漢字10,050字、非漢字1,183字の合計11,233文字を収録している。JIS X 0213では、漢字を4つのグループ（第1水準、第2水準、第3水準、第4水準）に分けて収録している。第1水準と第2水準は、0208の範囲と一致する。第3水準は、0212に含まれない漢字を中心に、新たに約3,600字が追加された。第4水準は、人名や地名、学術用語などに用いられる漢字約1,000字が追加された。

JIS X 0213の利用用途としては、多漢字を必要とする出版、印刷、行政システムなどが想定されている。

JIS2000とJIS2004との違い

JIS2004における最も重要な変更点は、**168文字の例示字形変更** と **10文字の追加登録** である。これらの変更は、文字の見た目、つまり字形に関わるもので、符号位置（コードポイント）自体の変更を伴うものとそうでないものがある。

例示字形の変更 (168文字)

この変更では、168文字の例示字形、つまり規格票で例示されているその文字の形が変更された。これらの文字は、符号位置は変更されておらず、表示される字形のみが変更された。

多くは、康熙字典体に基づく字形から、日本の常用漢字表・表外漢字字体表に準じた、より日本で一般的に使用される字体への変更である。例えば、「箸」の「はし」に点が付された他、「竹かんむり」の下の部分が「、」から「一」に変更されたことなどが挙げられる。

筆押さえの除去

叉、釜、茨、豹、箒、蛎、噌、曾、遡、溯、祷、禱、箆、筈、桧、檜、榧、罫、曙、蹟、蛯、蟯、噂、焔、鴎、鷗、迺、韮、薙、諌、勧、歎、竈、繋、繫、麹、屏、并、痩、蝋、藪

部首の分離・結合調整

蟹（解と虫を離す）、祟（出を小さく）、遥、瑶、揺、謡、遙、瑤、搖、謠、槙、枕、蒋、蔣、贋、巌、厳、嚴、欅、溜、榴、瑠、鱗、麟、璢、亙、渕、淵、猪、箋、煕、熙、桟、箭、笹、筏、筑、箪、篭、籠、竈、筐、箒、筥、簾、籐、籐、籟、蘯

一点しんにょうの形状変更

辻、辿、迄、逗、這、迚、迪、逖、迺、遉、適、遘、遨、遯、遶、遽、邂、邃、邇、邉、邊、邏

康熙字典体への変更

茨、豹、蝶、鐙、騨、曳、汎、氾、彙、謂、萎、畝、蝿、曖、蛯、謁、謔、鞄、甑、玩、顔、顏、巌、厳、嚴、翰、徽、祇、祗、嬉、竟、欣、欽、躯、駈、駆、勲、薫、醤、醬、嘱、囑、嘲、贖、醸、釀、蒋、蔣、壌、壤、濤、燈、灯、禰、祢、祷、禱、梅、楳、這、這、鉢、溌、醗、髪、髮、蛮、蠻、冶、弥、彌、躾、靡、麿、黙、默、籾、粐、籵、粃、粋、粹、粍、粁、糎、粧、糀、麭、麹、糒、糘、麪、麺、黴、黴、黶、靫、靭、靱、韭、韮、頚、頸、鶯、鴬、鷗、鴎、鵞、鵝、鵃、鴃、鷆、鷏、鷚、鴡、鵤、鷭、鶉、鶇、鵐、鷲、鴉、鵙、鵲、鶸、鶫、鶫、鶴、鵡、鵡、鶩、鷂、鷄、鶏、鶺、鶻、鶲、鵯、鷹、鷺、麒、麟

はい、承知いたしました。JIS X 0213:2004における10文字の追加登録に関する説明文と表を「だ・である」体に修正します。

JIS X 0213:2004における10文字の追加登録（字形追加）

JIS X 0213:2004の改正では、経済産業省の「文字情報基盤整備事業」の成果に基づき、印刷標準字体との整合性を高めることが目指された。この過程で、特に10組の漢字について、従来のJIS規格で採用されていた字形と印刷標準字体との間に差異があることが確認された。

この差異に対応するため、JIS X 0213:2004では、従来の字形を削除・変更するのではなく、それを維持したまま、新たに印刷標準字体として示された字形を追加で登録するという特別な措置が取られた。

例えば、「しかる」と読む文字では、従来のJIS字形「叱」(U+53F1) はそのまま従来の符号位置（面区点: 1-15-20）に残し、印刷標準字体である「叱」(U+20B9F) を新たな符号位置（面区点: 1-15-21）に追加した。

このように、10組の文字ペアそれぞれについて、従来の字形と印刷標準字体の両方がJIS規格内に存在し、利用者が文脈や目的に応じて選択できるようになった。この対応により、JIS規格は従来のデータとの互換性を保ちつつ、印刷標準字体やUnicodeとの整合性を向上させることに成功したのである。

以下の表は、JIS X 0213:2004で追加登録された10組の文字を示す。これは、従来の字形を置き換えるのではなく、印刷標準字体が追加されたことを示すものである。

従来のJIS字形	Unicode (従来)	追加された印刷標準字体	Unicode (追加)	備考
倶	`U+4FF1`	倶	`U+4FF1`	Unicode上は包摂
剥	`U+5256`	剝	`U+525D`	異なる符号位置
叱	`U+53F1`	叱	`U+20B9F`	異なる符号位置
呑	`U+541E`	呑	`U+541E`	Unicode上は包摂
嘘	`U+5653`	嘘	`U+5653`	Unicode上は包摂
妍	`U+59F8`	妍	`U+59F8`	Unicode上は包摂
屏	`U+5C4F`	屏	`U+5C4F`	Unicode上は包摂
并	`U+5E76`	幷	`U+5E7B`	異なる符号位置
痩	`U+75E9`	痩	`U+7626`	異なる符号位置
繋	`U+7E4B`	繋	`U+7E6B`	異なる符号位置

注:

- 表中の「Unicode」は、それぞれの字形に対応する代表的なUnicodeの符号位置を示す。
- 「Unicode上は包摂（統合）」とあるものは、Unicode規格ではこれらの字形を同一の符号位置に統合していることを意味する。JIS X 0213:2004では、これらに対しても異なる面区点コードを割り当てることで区別している。

字形変更の背景 - 表外漢字字体表、包摂基準、常用漢字表改定

JIS2004におけるこれらの字形変更は、単なるデザインの変更ではなく、以下のような複数の要因が複雑に絡み合った結果として行われた。

- **(1) 表外漢字字体表 (2000年) の影響:**
  JIS2000の制定当時、「表外漢字字体表」はまだ検討段階にあり、その内容はJIS2000には十分に反映されていなかった。表外漢字字体表では、康熙字典体を基にした「印刷標準字体」が示され、常用漢字表とは異なる字形が多く含まれていた。JIS2004では、この表外漢字字体表で示された「印刷標準字体」との整合性を図るため、多くの例示字形が変更された。これにより、表外漢字（常用漢字表に含まれない漢字）の字体が、より統一された形で表現されるようになった。

- **(2) 包摂基準の変更:**
  JIS2000では、漢字の字体のわずかな違いを「包摂」し、同じ符号位置に割り当てるという方針が採用されていた。例えば、「高」の「はしごだか」と「くちだか」は、同じ符号位置に割り当てられていた。しかし、この包摂基準が明確でなかったため、本来異なる字体として区別すべきものが包摂されている場合や、逆に、包摂すべきものが別の符号位置に割り当てられている場合があるという問題が指摘されていた。JIS2004では、この包摂基準が見直され、より厳密に適用されるようになった。その結果、168文字の例示字形変更と10文字の入れ替えが行われ、より正確な文字の区別が可能となった。

- **(3) 常用漢字表の改定 (2010年) の先取り:**
  2010年に改定された常用漢字表では、表外漢字字体表で示された「印刷標準字体」を参考に、一部の漢字の字形が変更された。JIS2004は、この常用漢字表の改定をある程度先取りする形で、常用漢字表に準じた字形への変更を行っている。例えば、「芦」や「塡」などの字形が、JIS2004で既に常用漢字表に準じた形に変更されている。ただし、完全な一致ではなく、2010年の常用漢字表改定時に追加で字形変更が行われた文字もある。

JIS2004における字形変更は、単なる見た目の問題ではなく、日本語の文字コードを取り巻く様々な状況変化に対応するための重要な改正であった。**表外漢字字体表**との整合、**包摂基準**の明確化、そして将来の**常用漢字表改定**への布石という、3つの大きな目的が背景にあった。この改正によって、日本語の文字表現がより正確で一貫性のあるものとなり、情報交換の円滑化に貢献することが期待された。JIS2004は、日本の文字コードの歴史において、大きな転換点となった規格といえる。

住民基本台帳ネットワーク統一文字

住民基本台帳ネットワーク統一文字は、市区町村の住民記録システムと都道府県、国を結ぶ住民基本台帳ネットワークシステム（住基ネット）で使用するために定められた文字集合である。

住民システムの縦割り解消とデータ連携の必要性

2000年代初頭、日本の行政システムは、各自治体が独自にシステムを構築・運用していた。住民に関する情報は各市区町村の住民基本台帳に保管されていたが、これらの情報は相互に連携されていなかった。例えば、転居した場合、転出元の自治体と転入先の自治体の両方で手続きが必要となり、国民にとって煩雑な手続きが伴った。

このような状況を打破し、行政サービスの効率化と国民の利便性向上を目指すため、全国の市区町村の住民基本台帳をネットワークで結び、本人確認情報の提供などを迅速に行うことを目的とした住基ネットの構築が計画された。この構想の実現には、全国の自治体で共通して利用できる文字コード体系の確立が不可欠であった。なぜなら、人名や住所といった個人情報を正確に連携するためには、使用する文字の定義を統一する必要があったからである。

住基ネット統一文字の必要性

住基ネットの構築にあたり、各市区町村が独自に管理していた文字コードの差異が大きな課題として浮上した。特に、地名・人名に用いられる漢字は多岐にわたり、自治体によっては独自の外字を作成・利用しているケースも少なくなかった。このような状況下では、異なる自治体間で正確な情報交換を行うことが困難であった。

そこで、総務省（当時の自治省）は、住基ネットで使用する統一的な文字集合として、**住基ネット統一文字**を策定した。これにより、全国の自治体で共通の文字基盤が確立され、円滑な情報連携が可能となった。

約2万字の文字セットと外字への対応

住基ネット統一文字は、運用開始当初のJIS X 0221（2001年4月20日制定版）のUCS-2に基づいて設計され、約2万字の文字が収録されている。この中には、JIS漢字やよく使われる人名漢字に加え、地名や歴史的な文献で使用される漢字、さらには主要なコンピュータベンダーが独自に拡張した文字なども含まれている。

主要ベンダーからの外字収集: 当時、各コンピュータメーカーは、自社のシステムで利用するために独自の外字セットを持っていた。住基ネット統一文字を策定するにあたり、これらの主要ベンダーから外字を収集し、共通の文字コードに含めることで、既存システムとの互換性を確保する努力がなされた。

画像による外字交換: しかし、人名漢字は非常に多種多様であり、約2万字の文字セットでも全てを網羅することはできなかった。そのため、住基ネット統一文字に含まれない文字（外字）については、「外字ファイル」と呼ばれる仕組みを用いて対応している。これは、文字コードに加えて、その文字のイメージデータ（画像）を一緒に登録することで、画面表示や印刷時に正確な文字を再現できるようにするものである。

UCSとの差異とハングル領域との競合

住基統一文字は、2002年8月5日から運用が開始された住民基本台帳ネットワークシステムで使用される文字体系である。当初のJIS X 0221（2001年4月20日制定版）のUCS-2に基づいて設計され、独自の拡張が施されている。現在の総文字数は21,170字であり、このうち漢字は19,435字を占めている。漢字の内訳は、JIS漢字13,141字、地名外字233字、メーカー拡張文字6,061字となっている。21,170字のうち、UCSと一致しているのは15,379字のみで、5,791字がUCSと異なる独自の実装となっている。特に深刻な問題として、追加漢字5,172字と変体仮名168字は、全てUCSのハングル領域と競合しており、ハングルと同時に使用することができない。この問題は現在も解消されていない。

この独自の実装は、当時の技術的な制約や、既存システムとの互換性を優先した結果であると考えられる。しかし、UCSとの非互換性は、異なるシステム間でのデータ交換において問題を引き起こす可能性があり、現在でも課題として残っている。

標準的な文字コードへの縮退問題

住基ネット統一文字で表現された文字を、JIS X 0208などの異なる文字コード体系のシステムで利用する場合、文字の縮退が発生する場合がある。特に、住基ネット統一文字にのみ含まれる文字は、JIS X 0208には存在しないため、適切に変換されないことがある。

環境によっては、この縮退処理がうまくいかず、本来の文字が代替文字である「●」（黒丸）や「〓」（ゲタ）に置き換えてしまう実装がある。これは、情報が正しく伝達されないという点で大きな問題となりうる。例えば、氏名や住所が「●●」と表示されてしまうと、誰の情報であるかを特定することが困難になる。

マイナンバー制度と住基ネット

2016年に導入されたマイナンバー制度においても、住基ネットは重要な役割を果たしている。マイナンバー（個人番号）は、住民票コードを変換して生成されるため、住基ネットの基盤の上に成り立っている。

また、マイナンバー制度における基本4情報（氏名、住所、生年月日、性別）の連携においても、住基ネットが重要な役割を担っている。各行政機関は、マイナンバーを識別子として、住基ネットに登録されている基本4情報を参照することで、国民に関する情報を正確に連携することができる。

住基ネット統一文字の現状と課題

住基ネット統一文字は、住基ネットの運用開始以来、大きな変更は行われていない。なお、住基ネットでは、住基ネット統一文字で同定できない氏名等の外字は、「外字ファイル」と呼ばれる特別なコードと画像のセットで管理・運用されている。これにより、文字コード上では表現できない外字も、システム内で一意に識別し、正確に扱うことが可能となっている。

住基ネット統一文字は、導入から年月が経過しており、情報技術の進展に伴い、いくつかの課題も顕在化している。UCSとの非互換性や、外字ファイルによる対応の煩雑さなどが挙げられる。近年、行政事務における情報システムの標準化が進められており、住基ネットにおいても、行政事務標準文字への対応が予定されている。これにより、異なる行政システム間での文字コードの互換性が向上し、さらなるデータ連携の円滑化が期待される。

戸籍統一文字

戸籍統一文字は、戸籍事務で使用するために定められた文字集合である。その名に反して、現在の戸籍システムで直接利用されているわけではない。戸籍法および戸籍法施行規則に基づき、法務大臣が告示によって定めた、戸籍事務における情報処理で使用可能な文字の範囲を示すものである。

戸籍事務のコンピュータ化が検討される中で、人名に用いられる多種多様な異体字の扱いが大きな課題となった。手書きで記録されてきた戸籍には、既存の文字コードでは表現できない異体字が多数存在したためである。

2002年（平成14年）の戸籍統一文字の制定は、このような背景のもと、戸籍情報を電子的に正確に記録・管理するための基盤を整備する目的で行われた。しかし、整備時点では全国民の戸籍情報を網羅的に把握していなかったため、法務省は、法令上利用可能と解釈できる文字を既存の

文字コードや文献などから網羅的に収集するという手法を採用した。このトップダウン的なアプローチの結果、戸籍統一文字には、実際に氏名として使用されていない文字も含まれることとなった。

また、戸籍の電算化の過程で、**改製不適合戸籍**という問題も生じた。これは、電算化の際に、文字コードの制約などにより、従来の戸籍に記載されていた情報が正確に移行できなかった場合に発生する。戸籍統一文字は、このような改製不適合戸籍における異体字の扱いを円滑にするための役割も担っている。

登記統一文字

登記統一文字は、登記事務で使用するために定められた文字集合であり、戸籍統一文字とは異なり、法務省が運営する登記情報システムで実際に利用されている。

登記事務においても、土地や建物の所有者名、会社名などに多くの異体字が用いられてきた。登記情報をコンピュータで管理する登記情報システムの構築にあたり、これらの異体字を正確に扱える文字コードが必要となった。

2004年（平成16年）に定められた登記統一文字は、戸籍統一文字を基盤としつつ、登記事務に特有の文字を追加したものである。法務省自身がシステムを運営しているため、実務で必要となる文字をより直接的に反映した文字集合となっている。

登記統一文字は、**戸籍統一文字**に、登記事務でのみ使用される**登記固有文字**を追加した構成となっている。

汎用電子情報交換環境整備プログラム

汎用電子情報交換環境整備プログラムは、2002年度から2008年度にかけて経済産業省の委託事業として実施された、電子政府実現のための文字情報基盤整備事業である。

1995年のWindows95発売以降、パソコンの一般家庭への普及とインターネットの急速な発展により、行政機関での文書電算化とインターネットを利用した電子申請システムの整備が進められた。この電子政府実現に向けて、住民の氏名や住所、法人の名称や所在地などを記載するための漢字の整備が不可欠となった背景があった。

実施体制と事業内容

本プログラムは、国立国語研究所、情報処理学会、日本規格協会の3機関が中心となり、内閣官房・総務省・法務省・文化庁が協力する5府省庁横断プロジェクトとして実施された。国立国語研究所は文字情報の整理・体系化、情報処理学会はデータベースの構築と運営、日本規格協会は統一的なデザインによる平成明朝体の制作を担当した。

成果物と課題

プログラムでは、戸籍統一文字と住民基本台帳ネットワーク統一文字の検討を行い、漢字情報データベースを構築した。また、登記統一文字の検討も実施された。しかし、その成果物である平成明朝体フォントは、出版・印刷・情報機器などの実業界からの協力を得て作成されたものの、権利関係の複雑さにより広く公開されることはなかった。

フォントの権利問題と後続事業

平成明朝体フォントの権利関係の複雑さにより、後の文字情報基盤事業では新たにIPAmj明朝フォントを開発する必要が生じた。特に、IVD（Ideographic Variation Database）登録の局面において、汎用電子情報交換環境整備プログラムと文字情報基盤事業の間でデザイン差についての対応が異なることとなり、課題を残す結果となった。

文字情報基盤

文字情報基盤整備事業の背景と歴史

文字情報基盤整備事業は、2010年度に経済産業省の電子経済産業省推進費により開始された、行政で使用される約6万文字の漢字を整備するプロジェクトである。この事業の前身として、2002年度から2008年度にかけて、国立国語研究所、情報処理学会、日本規格協会による「汎用電子情報交換環境整備プログラム」が実施された。

当初は経済産業省が主体となり、内閣官房、総務省、法務省、文化庁が参加する委員会で推進された。2011年4月以降は情報処理推進機構（IPA）に移管され、内閣官房IT担当室、経済産業省、IPAの3者による推進体制となった。その後、2020年8月に一般社団法人文字情報技術促進協議会に信託譲渡された。

住民システムへの普及には至らなかった文字情報基盤

事業では、IPAmj明朝フォントと呼ばれる約6万文字の漢字フォントを中心に、MJ文字情報一覧表や文字情報検索システム、文字情報基盤縮退マップなどが整備された。

自治体での導入については、IPAmj明朝フォントの完全導入か、既存システムとMJ文字図形との対応付けのいずれかの方法が提案されたものの、広く普及するには至らなかった。2013年6月に閣議決定された「世界最先端IT国家創造宣言」では、新規整備する情報システムにおいて文字情報基盤の活用を原則とすることが定められたが、システムベンダーの対応の遅れや導入時の技術的ハードルといった理由から、自治体間での情報連携における外字問題は現在も解決されていない。

行政事務標準文字

行政事務標準文字の概要

行政事務標準文字は、地方公共団体の情報システムにおける文字の標準化を目的として策定された文字セットである。地方公共団体情報システムの標準化に関する法律に基づき、基本方針において文字要件の標準化において、標準準拠システムへの移行にあたっては、自治体独自外字を行政事務標準文字に同定する義務が定められている。また、デジタル庁告示により整備中ベース・レジストリとして位置づけられている。

東日本大震災を契機とした戸籍副本の集約

2011年に文字情報基盤（MJ）が策定され、約6万字の文字セットが整備された。文字情報基盤文字を整備した当時は、住基ネット統一文字と戸籍統一文字で定義された文字セットを広く一般の情報システムで利用できるようにすることに主眼を置き、自治体で実際に氏名等として使われている外字についての網羅的で悉皆性のある調査は行われなかった。

2011年の東日本大震災で宮城県南三陸町、女川町、岩手県陸前高田市、大槌町の4市町の戸籍が正本・副本ともに津波で滅失したことを受け、法務省は2013年9月から戸籍副本データ管理システムの運用を開始し、全国2カ所の戸籍副本データ管理センターで副本データを一元管理する体制を整備した。

戸籍とマイナンバーの連携へ向けて文字情報を整備

その後、マイナンバー制度と戸籍制度を連携させるデジタル手続法が成立し、税と社会保障の一体改革に必要な戸籍情報をマイナンバーで活用できるよう法整備が行われた。この連携を実現するためには、マイナンバーと直接紐づく住民票コードを戸籍情報に連携させる必要があり、そのために氏名等の突合キーを用いた名寄せが必要となった。しかし、全国1,741自治体が戸籍情報連携システムを介して情報連携するにあたり、戸籍情報システムで使用されている文字が大量にあること、システムベンダーごとに文字の字形やコード体系が異なっていることが大きな課題となった。

この課題を解決するため、法務省は2018年から文字情報整備事業を開始し、全国の自治体の戸籍で使用されている約163万の文字を収集した。これらの文字について文字情報基盤との同定作業を実施し、約70万字に整理された後、さらに文字情報基盤に同定できた約55万字を除いた残りから、実際に戸籍で使用が確認された9,175字を特定した。この成果を活用し、デジタル庁はMJに収録されていない9,175文字をMJ+として追加して、約7万字からなる行政事務標準文字（MJ+）を定めた。

参照実装となるフォントの提供

デジタル庁は2024年春、行政事務標準文字の字形一覧やJIS X 0213への代替マップと合わせて、**追加文字行政事務標準明朝フォント**を自治体向けに提供した。このフォントをIPAmj明朝フォントと組み合わせて使うことで、行政事務標準文字を取り扱うことができる。

ところが自治体基幹システムで広く使われている帳票ミドルウェアをはじめとして、複数フォントの参照に対応していない業務アプリケーションが多いことから、単一ファイルでのフォントの提供を求める声が自治体・ベンダーから数多く寄せられた。

現時点ではOpenType形式のフォントファイルに収録できるグリフ数は65,536字という上限があることから、全ての行政事務標準文字を一つのフォントファイルに収めることは難しい。そこでデジタル庁では、行政事務標準文字のうち文字情報基盤文字に含まれる戸籍統一文字由来の、現時点では戸籍副本で利用を確認できていない文字を除いた約4万字を収録した**行政事務標準当用明朝フォント**を2024年9月末に提供開始した。

行政事務標準文字の今後

新たに追加した文字の国際標準化の必要性

2024年春に提供された追加文字行政事務標準明朝フォントのコードポイントは、暫定的に私用面（PUA）に置かれている。これらの文字を広く利用できるようにするためには、Unicodeとしての標準化が不可欠である。対応する基底文字が存在する場合は異体字としてIVDへの登録、他国に類似の文字が存在する場合はUnicodeの水平拡張としての登録、いずれにも該当しない場合は新たなコードポイントの登録という手順が必要となる。

同定できない登記固有文字や自治体独自外字の収録

現時点の行政事務標準文字は、戸籍副本システムで利用されている氏名等（氏名、本籍地）に使われている文字は全て収録済みである。しかしながら登記情報システム等で利用されている登記固有文字の多くは未収録である。登記固有文字の行政事務標準文字への同定は、令和6年度事業において作業を進めており、同定できなかった文字の取り扱いについて検討を要する。

今後は、自治体システムの標準準拠システムへの移行を通じて、自治体独自外字の文字同定が進展することが予想される。同定が困難な外字が生じた場合の収録方法、および戸籍副本では利用が確認できなかったものの自治体において利用が確認された文字の行政事務標準当用明朝フォントへの収録に向けた事務フローについて検討を進めている。

日本で現在も広く使われる各文字セットの関係

主要な文字セット間の包含関係を、ベン図を用いて視覚的に示すと、以下のようになる。

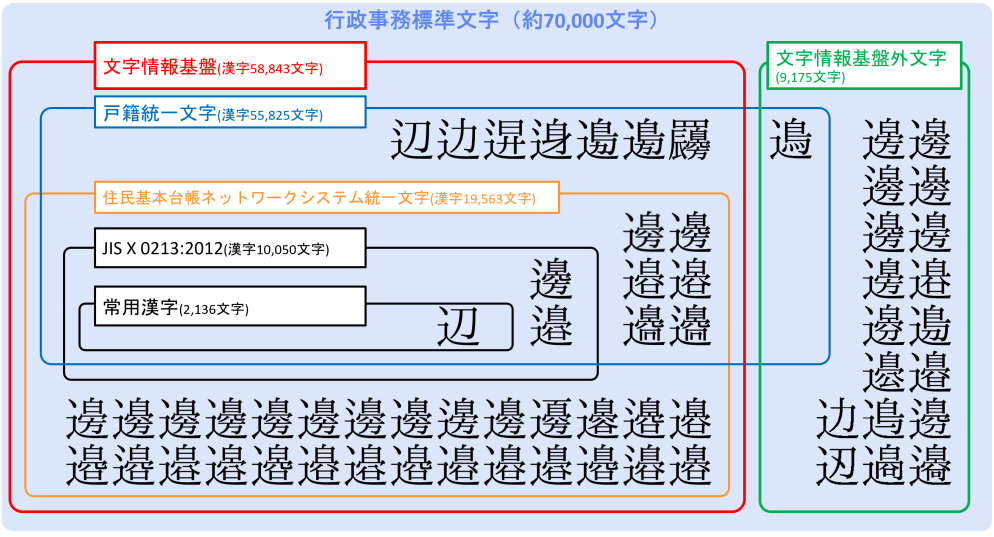


図1：主要な文字セットにおける「辺」の収録状況

以下は、日本で広く使われている主要な文字集合、文字コードの初出をまとめた表である。

文字集合・文字コード	文字数	初出	整備主体
JIS X 0201	191	1969年	日本規格協会
全銀テレ為替文字	158	1973年	全国銀行協会
教育漢字	1,026	1948年	文部科学省
常用漢字	2,136	1981年	文化審議会
人名用漢字	863	1951年	法務省
JIS X 0208	6,879	1990年	日本規格協会
JIS X 0212	12,946	1990年	日本規格協会
JIS X 0213	11,233	2000年	日本規格協会
入管正字	13,285	2012年	法務省入国管理局
住基ネット統一文字	19,563	1999年	総務省
戸籍統一文字	55,270	2004年	法務省
登記統一文字	68,067	2008年	法務省
文字情報基盤文字	59,325字	2011年	文字情報技術促進協議会
行政事務標準文字	69,839字	2023年	デジタル庁

各時代の主要文字セット、文字コードと年表

以下は、日本語文字コードの歴史的変遷における、各時代の主要な文字集合・文字コードと、関連する出来事をまとめた年表である。

年代	出来事	文字数
1963年	米国規格協会 **ASA X3.4**（後のANSI INCITS 4）制定	128
1967年	**ISO/IEC 646 制定**（各国別文字の予約）	128
1969年	**JIS C 6226-1969 制定**（漢字を追加）	約3,000
1974年	JIS C 6226-1974 制定	約3,000
1978年	**JIS C 6226-1978 制定**	約3,000
1981年	内閣告示第1号 **常用漢字表** 公布	1,945
1982年	アスキー（現KADOKAWA） **Shift_JIS**を開発	約6,800
1983年	**JIS X 0208-1983 制定** 常用漢字表に字体変更	6,877
1990年	JIS X 0208-1990 制定	6,879
1990年	JIS X 0212-1990 制定（**補助漢字**）	6,067
1991年	Unicode 1.0 公開（CJK統合漢字: 20,902）	7,161
1997年	JIS X 0208-1997 制定	6,879
2000年	JIS X 0213:2000 制定	11,233
2004年	**JIS X 0213:2004 制定** 常用漢字表を受け字体変更	11,233
2002年	**住民基本台帳ネットワークシステム全国稼働開始**	約20,000
2004年	戸籍法施行規則改正、**戸籍統一文字**	54,988
2009年	**登記統一文字** 策定	約60,000
2009年	汎用電子情報交換環境整備プログラム	約70,000
2011年	文字情報基盤整備事業 **IPAmj明朝フォント**	約60,000
2024年	**行政事務標準文字 策定**	約70,000

ユニコードの歴史

Unicodeとは、世界中のあらゆる文字を単一の文字コード体系で表現することを目的とした、現代のコンピューティングにおける基盤技術だ。本稿では、Unicodeの誕生背景から始まり、主要なバージョンにおける更新内容、その背景にある技術的・社会的な状況、そしてUnicodeを取り巻く論点や将来展望について詳しく解説していく。

Unicode登場の背景

コンピュータが登場した当初、文字を表現するためのコード体系は、それぞれの国や地域、あるいはコンピュータメーカーごとにバラバラに存在した。例えば、日本語を表示するために日本ではJISコードが、アメリカではASCIIコードが使用されていた。これにより、異なるコード体系を使用するコンピュータ間でデータのやり取りを行う際に、文字化けなどの問題が発生した。Unicodeは、このような状況を打開するために生まれた技術だ。

Unicode以前の文字コードは、テレタイプライターで使用されていた5ビットエンコーディングに起源を持つASCIIなど、限られた文字数しか表現できないものが主流だった。増加するコミュニケーションニーズに応えるために、より効率的なエンコーディングシステムが必要とされた。

Unicodeは、もともと、それまでに設計されたすべてのテキストエンコーディングに存在する制限を超えることを意図して設計された。各エンコーディングは、独自のコンテキストで使用するために依存していたが、他のエンコーディングとの互換性については特に期待されていなかった。実際、選択された2つのエンコーディングは、一緒に使用すると、多くの場合、まったく機能せず、一方のエンコーディングでエンコードされたテキストが、もう一方のエンコーディングでは文字化けとして解釈された。

根本的に、コンピュータは言葉や文字ではなく、数字を扱う。Unicode Standardは、ソフトウェアがデータを簡単に、破損のリスクを高くすることなく変換できるように、有用な文字を数字として記述している。一般的で統一された標準が登場する前は、コンピュータはテキストを変換するために独自のシステムを使用していた。そのため、情報が異なるコンピュータに渡されると、データが簡単に破損したり、不正確になったりする可能性があった。インターネットによって膨大な量のデータが大量のコンピュータによってやり取りできるようになると、標準の必要性は否定できなくなった。

このような背景から、1987年にXeroxのJoe Becker氏、AppleのLee Collins氏とMark Davis氏らによって、Unicodeの開発が始まった。Unicodeとは、「unique, unified, universal encoding（唯一の、統一された、普遍的なエンコーディング）」を意味し、世界中のあらゆる文字を単一のコード体系で表現することを目指した。1991年には、Unicodeの開発、拡張、普及を目的とした非営利団体であるUnicodeコンソーシアムが設立された。Unicodeコンソーシアムは、カリフォルニア州に設立され、Mark Davis氏が2023年まで初代会長を務めた。

Unicodeの登場により、異なる言語や文字体系を使用する人々が、コンピュータ上で互いにコミュニケーションをとることが容易になった。Unicodeは、インターネットの普及とグローバリゼーションの進展に大きく貢献し、現代のコンピューティングにおいて欠かせない存在となっている。

Unicodeのエンコーディング形式

初期のUCS-2とUCS-4

Unicodeの初期の設計においては、まず、世界中の文字に一意の符号位置を割り当てるUCS (Universal Coded Character Set) という概念が確立された。このUCSにおける符号位置を実際にコンピュータ上で表現するための符号化方式として、初期には固定長のエンコーディング方式が検討された。

その一つがUCS-2 (Universal Coded Character Set 2-byte form) である。これは、UCSで定義された符号位置を16ビット（2バイト）で表現する方式であり、約65,000字を表現することが可能であった。しかし、漢字を始めとする表意文字の増加や、古代文字、記号などの多様な文字を網羅する必要性が高まるにつれて、UCSで扱う文字数が当初の想定を大幅に超えることが明らかになった。16ビットでは表現しきれない符号位置が出現するという問題に直面し、この課題を解決するために、サロゲートペアという仕組みを導入したUTF-16が登場することになる。UTF-16は、UCS-2で表現できない符号位置を、2つの16ビットの組み合わせ（サロゲートペア）で表現する。

また、UCSの全ての符号位置を固定長の32ビット（4バイト）で表現するUCS-4 (Universal Coded Character Set 4-byte form) という符号化方式も考案された。UCS-4は、約40億の符号位置を表現でき、将来的に追加される可能性のある文字にも対応できるだけの十分な表現能力を持つ。UTF-32は、このUCS-4と実質的に同じエンコーディング方式であり、UCSの符号位置をそのまま32ビットで表現する。

これらの初期の符号化方式を経て、より柔軟で効率的な UTF (Unicode Transformation Format) エンコーディングが登場することになる。UTF-8、UTF-16、UTF-32は、いずれもUCSで定義された符号位置を異なる方法でバイト列に変換するエンコーディング方式である。

現在のUTF-8、UTF-16、UTF-32

現在のUnicode標準は、UTF-8、UTF-16、UTF-32の3つのエンコーディング方式を定義している。これらのうちUTF-8はASCIIとの後方互換性があるため、テキスト文書のファイル形式としては最も広く使用されている。UTF-16やUTF-32はUTF-8と比べて多くの文字を固定長で扱うことができ、より簡素に実装できることから、ソフトウェア内部で使用されることが多い。

- **UTF-8:** 可変長のエンコーディング形式で、1文字を1バイトから4バイトで表現する。ASCIIとの互換性があり、テキストファイルやWebページなどで広く使われている。
- **UTF-16:** 1文字を2バイトまたは4バイトで表現するエンコーディング形式である。JavaやWindowsの内部表現などで使われている。
- **UTF-32:** 大半のUnicodeコードポイントを4バイトで表現するエンコーディング形式である。処理は単純だが、データサイズが大きくなるデメリットがある。現代のUnicodeには後述する異体字セレクタや結合文字、組み文字、漢字構成記述文字などもあることから、表示上の全ての文字（書記素クラスタ）を4バイトの単一コードポイントで表現できる訳ではない。

Unicodeのバージョンと更新内容

Unicodeは、バージョンアップを重ねるごとに、新しい文字や記号が追加され、機能が拡張されてきた。Unicode 7.0以降、Unicode Technical Committeeはより予測可能なリリーススケジュールに従うことを決定した。標準の新しいメジャーバージョンは毎年リリースされる。Unicode 14.0以降、これらのリリースはそれぞれ第3四半期を目標としている。したがって、Unicode 14.0は2021年9月にリリースされ、Unicode 15.0は2022年9月にリリースされるなど、毎年新しいバージョンがリリースされている。

ここでは、主要なバージョンにおける更新内容とその背景について解説する。

Unicodeバージョンと追加情報

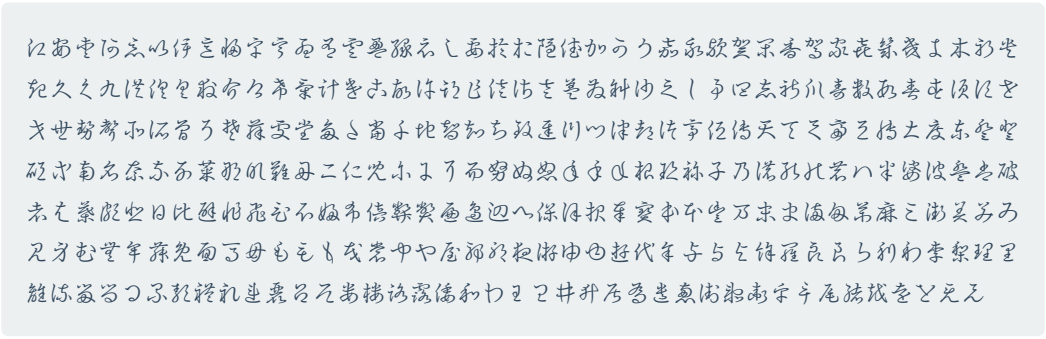
Unicode	発表年	追加された言語・機能	文字数
1.0	1991年	基本的なラテン文字、ギリシャ文字、キリル文字など。双方向テキストのサポート。	7,094
1.0.1	1992年	CJK統合漢字、CJK互換漢字	21,204
1.1	1993年	ハングル字母、ラテン文字拡張A、ギリシャ文字拡張。漢字の統一	5,969
1.1.5	1993年	ハングル字母、ラテン文字拡張追加、ギリシャ文字拡張	5,969
2.0	1996年	ハングル音節、サロゲートペア、CJK統合漢字拡張A	4,717
2.1	1998年	ユーロ記号、オブジェクト置換文字	2
3.0	1999年	シリア文字、ターナ文字、シンハラ文字、ミャンマー文字、エチオピア文字など	10,307
3.1	2001年	古代イタリア文字、ゴート文字、デザレット文字、ビザンチン音楽記号、音楽記号など	44,946
3.2	2002年	キリル文字補助、タガログ文字、ハヌノオ文字、ブヒッド文字、タグバンワ文字など	1,016
4.0	2003年	チベット文字拡張、統合カナダ先住民音節文字拡張、CJK統合漢字拡張Bなど	1,226
4.1	2005年	キリル文字補助、タガログ文字、ハヌノオ文字、ブヒッド文字、タグバンワ文字など。正規化	1,273
5.0	2006年	ンコ文字、バリ文字、ラテン文字拡張C、パスパ文字、フェニキア文字、楔形文字など	1,369
5.1	2008年	スンダ文字、オル・チキ文字、サマリア文字、ヴァイ文字、ブラーフミー文字など	2,817
5.2	2009年	キリル文字拡張A、ティフィナグ文字、リンブ文字など	38
6.0	2010年	ラテン文字拡張D、ラテン文字拡張E、ニューワ文字、タークリー文字、ソヨンボ文字など	2,088
6.1	2012年	エチオピア文字、インド系文字、記号など	684
6.2	2012年	キリル文字補助、その他の数学記号Aなど	41

Unicode	発表年	追加された言語・機能	文字数
6.3	2013年	ラテン文字拡張D、ミャオ文字、アラビア文字拡張Aなど	23
7.0	2014年	キリル文字拡張B、ラテン文字拡張F、追加の記号、絵文字など	2,849
8.0	2015年	矢印、ノーテーション記号、音楽記号、幾何学図形、装飾記号など	41
9.0	2016年	アドラム文字、装飾記号、絵文字など	72
10.0	2017年	古代ギリシャ文字、アラビア文字、カナダ先住民音節文字、カンナダ文字など	8,518
11.0	2018年	古代南アラビア文字、新南アラビア文字、フェストスの円盤文字など	684
12.0	2019年	絵文字、ラテン文字拡張Fのクリック音、アラビア文字拡張Cのアラビア文字、仮名拡張Aの平仮名と片仮名の二重音字など	554
12.1	2019年	グルジア語拡張、ハニフィ・ロヒンギャ語、古代ソグド語、ソグド語、ドグラ語など	62
13.0	2020年	線文字B表意文字、線文字B表意文字補助、キプロス音節文字など	5,930
14.0	2021年	絵文字、動物、食べ物、衣服、アクティビティなど	3,058
15.0	2022年	新しいブラーフミー文字、新しいカナダ先住民文字、キプロス文字の新しい文字、追加の音楽記号、トランプの記号など	4,489
16.0	2023年	古代ペルミ文字、コリヤーク語、契丹小字、契丹大字、タンサ語、カヤーリ語など	4,192

Unicodeのバージョンアップは、単に文字を追加するだけでなく、文字コードを取り巻く技術的・社会的な状況の変化に対応するために、様々な改良や修正が行われてきた。例えば、Unicode 2.0ではサロゲートペアの導入により、表現できる文字数が大幅に増加した。Unicode 4.1ではモンゴル文字（ウイグル文字で書かれたモンゴル語）を表記するためにIVS/IVDが導入され、その後は漢字の異体字を扱うための仕組みとしても活用されるようになった。また、Unicode 6.0では絵文字が追加され、コミュニケーションの手段として多様化が進んだ。

Unicodeへの変体仮名の収録

Unicodeに収録されている変体仮名は286文字で、以下の通りである。



これらの変体仮名は、追加多言語面（面01）ではなく、追加漢字面（面02）に配置されている。また、「濁点・半濁点付き変体仮名テスト用チャート」に示された濁音および半濁音の変体仮名も存在する。これらの文字コードは、清音の変体仮名の文字コードに濁点もしくは半濁点の文字コードを組み合わせることで表現される。

濁点・半濁点付き変体仮名テスト用チャート

濁点付き変体仮名(144グリフ)、半濁点付き変体仮名(36グリフ)のグリフを追加実装した。
（IPAmj明朝フォントを正しく指定しても、一部の環境では合成グリフが正しく表示されない場合があります）

#	ベース変体仮名UCS	ベース変体仮名	Combining sound mark	変体仮名合成結果	音価	仮名合成結果
1	1B017	𛀗	゙	𛀗゙	か	が
2	1B018	𛀘	゙	𛀘゙	か	が
3	1B019	𛀙	゙	𛀙゙	か	が
4	1B01A	𛀚	゙	𛀚゙	か	が
5	1B01B	𛀛	゙	𛀛゙	か	が
6	1B022	𛀢	゙	𛀢゙	か	が
7	1B01C	𛀜	゙	𛀜゙	か	が
8	1B01D	𛀝	゙	𛀝゙	か	が
9	1B01E	𛀞	゙	𛀞゙	か	が
10	1B01F	𛀟	゙	𛀟゙	か	が
11	1B020	𛀠	゙	𛀠゙	か	が
12	1B021	𛀡	゙	𛀡゙	か	が

図1：濁点・半濁点付き変体仮名テスト用チャート（「が」の部分）

Unicodeへの収録にあたっては、まず既存の文字データベースやフォントなどを調査し、収録する変体仮名の文字数を特定した。

Unicodeに変体仮名が収録された経緯

変体仮名は、平仮名の異体字であり、現代の日常生活ではほとんど用いられていない。しかし、1947年以前の戸籍には変体仮名で記載されたものが多く存在し、戸籍行政では約200年間、変体仮名をサポートする必要がある。また、歴史的な文献にも変体仮名で記述されたものが多く存在する。これらの文書をコンピュータで処理するためには、変体仮名を文字コードとして符号化することが必要となる。

変体仮名はUnicode 10.0より前に、文字コード標準（JIS規格やISO規格）に収録されていなかった。しかし、メーンフレームの時代から、国内ベンダー各社は変体仮名を独自に符号化していた。そのため、変体仮名を含む文書を異なるシステム間で交換する場合、文字化けなどの問題が発生することがあった。

このような状況を改善するため、2015年に変体仮名文字コード標準化の提案が行われた。この提案は、特定分野の文字情報交換における記述性の向上を目指したものであった。その結果、Unicode 10.0で変体仮名が収録されることとなった。

Unicodeへの変体仮名の登録時に発生した議論

Unicodeへの変体仮名の登録時には、主に文字セットの選定と包摂という2つの観点から議論があった。

変体仮名文字セットの選定

変体仮名は、同じ音価を表すものでも、字形が異なるものが多数存在する。そのため、Unicodeに収録する変体仮名の文字セットを選定する際には、どの字形を収録するのか、という問題が発生した。

この問題に対しては、活版印刷やデジタルフォントから集字し、学術情報交換用変体仮名セットを選定するという方法がとられた。このセットには、変体仮名の機能的使い分けを表現するため、同字母異体も収録されている。学術情報交換用変体仮名セットは、戸籍行政における変体仮名の使用状況も考慮して選定された。

変体仮名の包摂

変体仮名は平仮名の異体字であるため、既存の平仮名との関係をどのように整理するのかという問題が発生した。

一般に、文字コード標準化のための文字セットは、解釈に揺れのない安定した集合で、かつ、多数の使用者が合意できる集合でなくてはならない。しかし、変体仮名は、同じ音価を表すものでも多様な字形が存在し、その解釈や使用範囲は時代や地域、個人によって異なる場合がある。このような変体仮名の性質は、安定した文字セットの構築というUnicodeの要件と相反する側面があった。

Unicodeへの変体仮名の登録時における包摂の問題は、変体仮名を平仮名とは別の文字として扱うことで整理された。これは、変体仮名と平仮名の間には、字形の違いだけでなく、歴史的な背景や使用される文脈の違いもあるためである。

具体的には変体仮名はUnicodeの追加漢字面（面02）に配置され、平仮名とは異なる符号位置に割り当てられている。これにより、変体仮名と平仮名を明確に区別することが可能となった。

「令和」改元時の元号合字の追加

新元号「令和」の合字「㋿」のUnicodeへの収録は、以下のような経緯で進められた。
2017年6月16日に「天皇の退位等に関する皇室典範特例法」が公布され、2017年12月13日に2019年4月30日を施行期日とする政令が公布された。Unicode Consortiumは2018年1月25日の時点で、新元号の合字のためにコードポイントU+32FFを予約していた。2019年4月1日の新元号発表後、Unicode 12.1として「令和」の合字が正式にリリースされ、総文字数は137,929となった。合字のコードポイントU+32FFは、明治(㍾)、大正(㍽)、昭和(㍼)、平成(㍻)といった従来の元号合字と同様の扱いとなった。

合字の必要性を巡る議論

現代のワープロソフトでは漢字2文字の横幅を狭めて1文字分の空間に押し込むことが容易になっており、合字の必要性自体に疑問が投げかけられていた。特に、合字を使用することを前提とした古いシステムがUnicodeを利用しているかという技術的な懸念も存在していた。しかし、従来の元号との一貫性や互換性維持の観点から、新元号でも合字が実装された。

㋿への主要ベンダーの対応

主要ベンダー各社は2019年の改元に合わせて令和の合字（U+32FF）への対応を進めた。アドビは4月11日までに小塚明朝、小塚ゴシック、源ノ角ゴシックなどの日本語フォントおよびPan-CJKフォントのアップデートを完了した。マイクロソフトは5月1日の改元に合わせてWindowsの各バージョンに更新プログラムを提供したが、MS932（シフトJIS）での対応は見送られた。アップルはUnicode 12.1のリリースに合わせてmacOSのシステムフォントとして令和合字を実

装し、各種フォント形式での表示に対応した。各ベンダーは明治(㍾)、大正(㍽)、昭和(㍼)、平成(㍻)と同様の扱いで実装を行い、2019年5月までには主要なプラットフォームでの対応が完了した。

Unicodeを取り巻く論点

Unicodeは、世界中の文字を統一的に扱うための重要な基盤技術だが、その一方で、いくつかの論点も存在する。

セキュリティに関する課題

Unicodeは、非常に多くの文字を収録しているため、その中には悪意のある利用に繋がる可能性のある文字も存在する。例えば、見た目が似ている文字を悪用して、ユーザーを騙したり、システムに侵入したりする攻撃が考えられる。Unicodeを使用する際には、このようなセキュリティリスクを考慮する必要がある。

正しい実装の難しさ

Unicodeは複雑な仕様であるため、ソフトウェア開発者がUnicodeを正しく実装することは容易ではない。文字列の長さの計算、文字の結合、正規化などの処理を適切に行わないと、文字化けやデータの破損などの問題が発生する可能性がある。

データベースにおける課題

Unicodeのコードポイントをそのままデータベースに格納する場合、テキストデータのソート方法が課題となる。Unicodeでは、Unicode照合と呼ばれるコードポイントのソート順序を定義しているが、データベースにおける照合のサポートは、まだ十分ではない。また、言語によっては、単語や文字のソート順序について合意が得られていない場合があり、標準的な照合の確立が課題となっている。

既存のエンコーディングとの互換性

Unicodeは、既存の文字エンコーディングとの互換性を維持するために、多くの妥協を強いられてきた。既存のエンコーディングとの完全な「ラウンドトリップエンコーディング」を維持できることが絶対に重要だ。そうしないと、エンコーディングの混乱を一掃することはできない。その結果、Unicodeには、一部の専門家から見て疑問視されるような文字や設計上の選択が含まれているという指摘もある。

Unicodeの将来展望

Unicodeは、今後も進化を続け、より多くの文字や記号を収録し、機能を拡張していくことが予想される。Unicodeコンソーシアムは、絶滅危惧言語や先住民族の言語の保護にも力を入れており、これらの言語をUnicodeに収録することで、デジタル世界における言語の多様性を維持することを目指している。

また、Unicodeは、拡張現実 (AR) や仮想現実 (VR) 、機械学習などの新しい技術にも対応していく必要がある。これらの技術の発展に伴い、Unicodeは、より複雑な文字や記号を表現する必要性が出てくる可能性がある。

Unicodeの将来は、技術的な進化だけでなく、社会的なニーズの変化にも大きく影響される。グローバリゼーションの進展、多言語化の進展、文化的多様性の尊重など、Unicodeを取り巻く社会的な状況の変化に対応していくことが、Unicodeの将来にとって重要だ。

Unicodeと関連標準の開発において進行中のプロジェクトは多数ある。Unicodeエンコーディングには、新しい文字とスクリプトが追加され続けている。また、Unicode Character Databaseとそれに関連する標準では、プロパティとアルゴリズムが追加されている。

Unicode Technical Standard (UTS) #55「Unicode Source Code Handling」は、ソースコードにおける文字の偽装攻撃を防ぐための新しい技術仕様である。この仕様は、キリル文字やラテン文字など、視覚的に区別が困難な文字を使用した識別子の偽装や、双方向テキストの制御文字を悪用した攻撃（Trojan Source）などへの対策を定めている。

2024年1月に公開された第2版では、プログラミング言語設計者や開発環境開発者向けのガイダンスが提供され、ICUライブラリを通じて実装されている。また、UAX #9（双方向アルゴリズム）やUAX #31（識別子と構文）など、関連する仕様も同時に更新された。

Unicodeへ文字の追加

Unicodeは、現代の文字だけでなく、古代の文字や歴史的な文字、数学記号、絵文字なども含んでいる。いわば、Unicodeは「文字とコードポイントの対応表」のようなものであり、世界中の文字に固有の番号を割り当てることで、コンピュータでの文字処理を可能にしている。

Unicodeコンソーシアム

Unicodeの開発・管理を行っているのは、Unicodeコンソーシアムという非営利団体である。Unicodeコンソーシアムは、Apple、IBM、Microsoft、Googleなど、世界の主要なIT企業や学術機関が参加しており、Unicode規格の策定、改訂、普及活動などを行っている5。

Unicodeへの文字追加フロー

Unicodeへの文字追加は、誰でも提案することができる。新しい文字の追加を希望する個人や団体は、Unicodeコンソーシアムに提案書を提出する必要がある。提案書には、文字の形状、読み方、用途、使用されている地域・文化圏などの情報に加え、文字の必要性や既存の文字との違いを明確に示す必要がある。

Unicodeへの文字追加は、以下のフローに従って行われる。

1. **提案**: 新しい文字の追加を提案する。提案は、Unicodeコンソーシアムのウェブサイトから提出することができる。提案書には、文字に関する詳細な情報と、その文字が必要である理由を記述する必要がある。
2. **評価**: Unicodeコンソーシアムの専門家委員会が、提案された文字を評価する。評価基準には、文字の必要性、既存の文字との重複、文字の安定性などが含まれる1。専門家委員会は、必要に応じて追加情報を要求したり、提案者に修正を依頼したりする。
3. **承認**: 専門家委員会の評価に基づき、Unicode技術委員会が文字の追加を承認する。承認された文字は、Unicode規格に追加されることが決定する。
4. **登録**: 承認された文字は、Unicode文字データベースに登録される。Unicode文字データベースは、Unicodeに含まれるすべての文字と、そのコードポイント、文字名などの情報を格納したデータベースである。
5. **公開**: 新しいUnicode規格の一部として、追加された文字が公開される。Unicode規格は、定期的に改訂され、新しい文字が追加される。

Unicodeに文字が収録される基準

Unicodeに文字が収録される基準は、以下の通りである。

- **明確な識別性**: 他の文字と明確に区別できること。
- **実用性**: 実際に使用されている、または使用される可能性が高いこと。
- **安定性**: 文字の形状や意味が安定していること。
- **非重複性**: 既存のUnicode文字と重複しないこと。

Unicodeの構造

Unicodeは、17の面に分割されている。各面には、65,536個のコードポイントを割り当てることができる。

面 / 符号位置	名称	収録されている主な文字
第0面 (U+0000 - U+FFFF)	BMP: Basic Multilingual Plane 基本多言語面	基本的な文字
第1面 (U+10000 - U+1FFFF)	SMP: Supplementary Multilingual Plane 追加多言語面	古代文字や記号・絵文字類など
第2面 (U+20000 - U+2FFFF)	SIP: Supplementary Ideographic Plane 追加漢字面	漢字専用領域
第3面 (U+30000 - U+3FFFF)	TIP: Tertiary Ideographic Plane 第三漢字面	追加漢字面に入りきらなかった漢字。また、将来的には古代漢字や甲骨文字などが収録される予定
第4面 - 第13面 (U+40000 - U+DFFFF)	/ /	未使用（将来どのような目的で使用するのかすら決まっていない）
第14面 (U+E0000 - U+EFFFF)	SSP: Supplementary Special-purpose Plane 追加特殊用途面	制御コード専用領域
第15面 - 第16面 (U+F0000 - U+10FFFF)	PUP: Private Use Plane 私用面	BMPの U+E000 - U+F8FF の領域の拡張

CJK統合漢字における字形を越えた包摂

各国の漢字において、Unicodeに収録されるものと収録されていないものがあるのは、以下の理由による。

- **漢字統合**: Unicodeでは、微細な差異はあっても本質的に同じ文字であれば、一つの番号を当てる方針で各国・各社の文字コードの統合を図っている 1。これを漢字統合（Han Unification）という。漢字統合は、中国、日本、韓国、台湾など、漢字を使用する国々の間で、共通の文字コード体系を構築するために実施された。しかし、漢字統合によって、各国で独自に発展してきた漢字文化や、漢字の字体、意味、用法などの違いが失われる可能性があるという批判もある。
- **文字の必要性**: Unicodeに収録される文字は、世界中で使用される文字であることが求められる。そのため、特定の国や地域でのみ使用される漢字は、Unicodeに収録されない場合がある。
- **文字の安定性**: Unicodeに収録される文字は、形状や意味が安定している必要がある。そのため、歴史的な文献にのみ出現する漢字などは、Unicodeに収録されない場合がある。

CJK統合漢字のベースとなった主な文字集合は、中国（C）の GB 2312、台湾（C）の CNS1,2,14面、日本（J）の JIS X 0208 と JIS X 0212 、韓国（K）の KS C 5601 である。これらの文字集合を統合する際、字形的に同一もしくは些細な字形差と認められる文字は、同じコードポイントに割り振ることで、Unicode 1.0.1 時点で 20,902 という数にまとめられた。

漢字を追加する場合の方法

漢字を追加する場合、以下の3つの方法がある。

- **コードポイントの追加**: 新しいコードポイントを割り当てる方法。Unicodeの文字領域に空きがある場合に用いられる。
- **水平拡張**: 既存の文字コード規格に新しい漢字を追加し、Unicodeと対応付ける方法。日本の場合は、JIS X 0213の改訂などがこれにあたる。
- **IVD登録**: 異体字セレクタ (IVS) を用いて、既存の漢字の異体字として登録する方法。IVSは、漢字の微妙な字体差を記述するための仕組みである。

UnicodeとCJK統合漢字

Unicodeにおける漢字の扱いは、CJK統合漢字を中心に展開されている。CJK統合漢字は、中国、日本、韓国で共通して使用される漢字を統合したものであり、Unicodeの主要な構成要素の一つである。CJK統合漢字は、CJK-JRG（中国・日本・韓国の共同作業部会）において作業が行われた。

CJK統合漢字は、Unicodeの第0面（基本多言語面）のU+4E00からU+9FFFに収録されている。CJK統合漢字には、約2万字の漢字が収録されており、中国、日本、韓国のいずれの言語でも使用することができる。

課題と展望

Unicodeは、世界中の文字を統一的に扱うための重要な規格であるが、いくつかの課題も抱えている。

- **漢字統合による文化的な影響**: 漢字統合は、異なる文化圏で異なる意味や用法を持つ漢字を同一視してしまう可能性があり、文化的な多様性を損なう可能性があるという指摘がある 11。
- **IVSの普及**: IVSは、漢字の微妙な字体差を表現するための有効な手段であるが、IVSに対応したフォントやアプリケーションはまだ十分に普及していない。IVSの普及を促進するためには、フォントベンダーやアプリケーション開発者への働きかけが必要である。
- **新しい文字の追加**: Unicodeは、常に進化し続けており、新しい文字が追加されている。しかし、新しい文字を追加する際には、既存の文字との整合性や、文字の安定性などを考慮する必要がある。

まとめ

Unicodeは、世界中の文字を統一的に扱うための重要な基盤技術であり、インターネットの普及やグローバリゼーションの進展に大きく貢献してきた。Unicodeが登場したことにより、言語の壁を越えたコミュニケーションが可能になり、情報共有が促進された。

しかし、Unicodeは、単なる技術的な標準ではない。Unicodeは、文化的多様性を尊重し、世界中の人々がデジタル世界で平等にアクセスし、参加できるための基盤でもある。Unicodeの進化は、技術的な進歩だけでなく、社会的なニーズや価値観の変化を反映したものであり、今後も、多様性と包容性を重視した方向に進展していくことが期待される。

ユニコードの技術仕様

インターネットの普及により、世界中の人々が情報を共有するようになった。そのため、多言語に対応した統一的な文字コードの必要性が高まった。このニーズに応える形で登場したのが**Unicode**である。

従来の文字コードは、特定の言語や地域での利用を前提としていたため、異なる文字コード間でのデータ交換には、文字化けなどの問題が発生していた。インターネット時代を迎え、世界中の人々が情報を共有するためには、あらゆる言語を統一的に扱える文字コードが必要とされた。

Unicodeは、世界中のすべての文字を単一の文字集合で表現することを目指して開発された国際的な文字コード規格である。Unicodeの登場により、多言語環境での文字化け問題は大幅に改善された。

Unicodeの基本的な仕組み

Unicodeでは、各文字に固有の数値（**コードポイント**）を割り当てる。コードポイントは、通常、`U+`に続けて4桁から6桁の16進数で表記される（例：`U+4E00`、`U+20BB7`）。

符号位置 (Code Point)

コードポイントとは、Unicodeの文字集合における文字の位置を示す番号である。Unicodeでは、文字集合全体を、**基本多言語面**（BMP）と**追加面**に分けて管理している。BMPは、U+0000からU+FFFFまでの範囲で、よく使われる文字が配置されている。追加面は、U+10000以降の範囲で、BMPに入りきらなかった文字や、特殊な用途の文字が配置されている。

符号化方式 (Encoding)

Unicodeのコードポイントを、実際にコンピュータで扱うデータ（バイト列）として表現する方法を、**エンコーディング**（符号化方式）と呼ぶ。主なエンコーディング方式として、**UTF-8**、**UTF-16**、**UTF-32**がある。

- **UTF-8**: 1文字を1バイトから4バイトの可変長で表現する。ASCII文字は1バイトで表現されるため、ASCIIとの互換性が高い。インターネット上で最も広く用いられているエンコーディング方式である。

- **UTF-16**: 1文字を2バイトまたは4バイトで表現する。BMP内の文字は2バイト、追加面の文字は4バイト（サロゲートペア）で表現される。WindowsやJavaなどで用いられている。
- **UTF-32**: 1文字を4バイトの固定長で表現する。すべての文字を同じ長さで表現できるため、処理は単純になるが、データサイズは大きくなる。

包摂規準（Unification）

Unicodeでは、異なる言語で用いられる同じ字形の漢字を、原則として1つのコードポイントに統合する**包摂規準**（Unification）という考え方を採用している。例えば、日本語の「日」、中国語の「日」、韓国語の「日」は、同じ字形であれば、すべて`U+65E5`という1つのコードポイントに統合される。

この包摂規準により、文字数を大幅に削減し、効率的なデータ管理を可能にしている。しかし、一方で、異なる言語で微妙に字形が異なる漢字も統合されてしまうため、問題が生じる場合もある。

異体字問題の顕在化

包摂規準は、効率性と引き換えに、異体字の問題を顕在化させた。例えば、「葛」と「葛」は、日本では異なる文字として扱われるが、Unicodeでは同じ「葛」に包摂されてしまう。そのため、これらの異体字を区別して表現するためには、後述するIVSなどの技術を用いる必要がある。

CJK統合漢字

Unicodeでは、中国語（C）、日本語（J）、韓国語（K）で用いられる漢字を、**CJK統合漢字**としてまとめて扱っている。CJK統合漢字は、Unicodeの大きな部分を占めており、その数は数万字に及ぶ。

統合のメリットと課題

CJK統合漢字のメリットは、文字数を大幅に削減できることである。例えば、日本語のJIS X 0208と中国語のGB2312に含まれる漢字を単純に合計すると、約1万3千字になるが、CJK統合漢字では約7千字に統合されている。

一方、CJK統合漢字の課題は、包摂規準によって異なる言語で、微妙に字形が異なる漢字が統合されてしまうことである。例えば「骨」という漢字は、日本（骨）と中国（骨）、台湾（骨）では字形がわずかに異なるが、Unicodeでは同じコードポイントに統合されている。

各漢字表に含まれる漢字のマッピング

CJK統合漢字は、各国の漢字表を基に策定されている。例えば、JIS X 0208、JIS X 0212、JIS X 0213、GB2312、GB18030、Big5、KS X 1001などの漢字表に含まれる漢字が、UnicodeのCJK統合漢字にマッピングされている。

サロゲートペア

サロゲートペア (Surrogate Pairs)は、Unicodeにおいて、基本多言語面（BMP：Basic Multilingual Plane、U+0000からU+FFFF）の外に配置された文字（追加面）を表現するための仕組みである。

Unicodeの基本多言語面には、65,536（= 2^{16}）文字分の符号空間が用意されている。しかし、漢字などを含む多くの文字を収録するためには、この空間だけでは不足する。そこで、Unicodeでは、U+10000以降の符号空間を「追加面」として定義し、基本多言語面には収まらない文字を配置している。

サロゲートペアでは、基本多言語面のうち、**上位サロゲート**（U+D800からU+DBFFの範囲、1,024文字分）と**下位サロゲート**（U+DC00からU+DFFFの範囲、1,024文字分）と呼ばれる特殊な領域を組み合わせて使用する。具体的には、上位サロゲートと下位サロゲートのペア（2文字）で、1つの追加面の文字を表現する。

例えば、追加面に含まれる「𪜈」（ほっけ、U+29E3D）という漢字は、UTF-16では、上位サロゲートの「D867」（U+D867）と下位サロゲートの「DE3D」（U+DE3D）の2文字で表現される。

行政事務における留意点: 行政事務で扱う人名や地名には、追加面に配置された漢字が含まれる場合がある。システム開発やデータ入力の際には、サロゲートペアを正しく処理できることを確認する必要がある。

IVS: 異体字シーケンス

異体字シーケンス（Ideographic Variation Sequence）は、漢字の異体字を表現するための仕組みである。**異体字**とは、標準とされる字体に対して、字形が異なるが同じ文字と見なされる文字のことである。例えば、「辺」と「邊」、「斉」と「齊」、「高」と「髙」などは、異体字の関係にある。

IVSでは、**基底文字**（base character）と**異体字セレクタ**（variation selector）と呼ばれる特殊な制御文字を組み合わせて、異体字を表現する。基底文字は、異体字関係にある文字の代表となる文字（例えば、「辺」）であり、異体字セレクタは、基底文字に後続して、その異体字を具体的に指定する（例えば、「邊」や「邉」）。

異体字セレクタには、**標準化された異体字セレクタ**（Standardized Variation Selectors：SVS）と、**漢字異体字データベース**（Ideographic Variation Database：IVD）に登録された異体字セレクタの2種類がある。

SVSとIVSの区別

標準化された異体字セレクタ（SVS）は、非漢字やCJK互換漢字などで利用され、基本多言語面（BMP）のU+FE00〜U+FE0Fの範囲に定義されている[1][2]。

一方、**漢字異体字シーケンス（IVS）**は漢字専用で、追加特殊用途面（SSP）に定義された異体字セレクタ（U+E0100〜U+E01EF, VS17〜VS256）を使用する[4]。

IVDの登録コレクション

漢字異体字データベース（IVD）には、現在3つの主要なコレクションが登録されている：

- **Adobe-Japan1**：Adobe社が管理するAdobe-Japan1グリフ集合のコレクション
- **Hanyo-Denshi**：汎用電子情報交換環境整備プログラム委員会が管理する、住基ネット・戸籍システム・登記システム等での利用を想定したコレクション
- **Moji_Joho**：経済産業省の汎用電子情報交換環境整備プログラムの成果を引き継ぎ、IPAmj明朝フォントの基となるコレクション

なお、同じ字形であってもAdobe-Japan1とHanyo-Denshi（Moji_Joho）では異なる異体字セレクタとして登録されている場合があるため、使用時には注意が必要である。

例えば、Adobe-Japan1コレクションで「葛」という異体字を表現する場合、基底文字「葛」（U+845B）に、IVD異体字セレクタ「E0100」を後続させて、「葛」（U+845B U+E0100）と表現する。これに対してMoji_Johoコレクションでは「葛」（U+845B U+E0102）と表現する。

IDC: 漢字構成記述文字

漢字構成記述文字（Ideographic Description Characters）は、Unicodeにおいて、漢字の構造を記述するための特殊な文字体系である1。この仕組みは、特に未定義の漢字や表示不可能な漢字を、既存の部品を用いて論理的に表現することを可能にする。

IDCは基本多言語面のU+2FF0からU+2FFFの範囲に配置されており、現在12個の記述文字が定義されている。これらの文字は、漢字の構成要素がどのように配置されるかを示す前置演算子として機能する。

IDCの構造と使用法

IDCを用いた漢字の記述は、**演算子**（漢字の構成要素の配置方法を指定する特殊文字）と、**オペランド**（実際の漢字や部首などの構成要素）によって構成される。これらの組み合わせによって作られる文字列を漢字構成記述文字列（Ideographic Description Sequence: IDS）と呼ぶ。

IDCの主要な演算子

IDCには以下のような基本的な演算子が含まれる：

- ⿰（U+2FF0）：左右の構成要素を結合
- ⿱（U+2FF1）：上下の構成要素を結合
- ⿲（U+2FF2）：3つの要素を横に結合
- ⿳（U+2FF3）：3つの要素を縦に結合
- ⿴（U+2FF4）：外枠内に要素を配置

IDCによる漢字の構造記述の具体例を、下記にいくつか例示する。

漢字	IDC構造記述	説明
明	⿰日月	左右に分割
林	⿰木木	左右に同じ要素を配置
海	⿰氵每	左に水部首、右に每
謝	⿰言射	左に言部首、右に射
品	⿱⿰口口口	上下構造と左右構造の組み合わせ
鬱	⿴冖⿱穴⿰冊冫⿱⿰幺言幺⿰長馬長刂心	複雑な入れ子構造

IDC記述における技術的な留意点

- IDCによる記述は可変長となり、文字によって長さが大きく異なる
- 同じ文字でも複数の異なるIDC表現が可能である
- IDCを処理できるシステムは限定的である
- Unicode等の文字コードでの表現と、IDCでの表現が一致しない場合がある

IDC記述の実装上の課題

- IDCによる表現は、文字コード化されていない漢字の記述には有用である
- IDCの解釈や処理には専用のソフトウェアが必要となる
- 一つの漢字に対して複数の異なるIDC表現が存在する可能性があり、正規化が困難である

Unicodeにおける可変長の文字表現

サロゲートペアやIVSを使用すると、文字の表現が可変長になる。以下では、具体的な文字を例に、この可変長の文字表現について解説する。

文字符号化方式によるビット長の違い

まず、代表的な文字について、UTF-8, UTF-16, UTF-32での表現を見てみよう。それぞれのエンコーディングにおけるビット長の違いを直感的に理解することができる。

文字	符号位置	UTF-8	UTF-16	UTF-32
A	U+0041	41 (1)	0041 (2)	00000041 (4)
日	U+65E5	E6 97 A5 (3)	65E5 (2)	000065E5 (4)
𩸽	U+29E3D	F0 A9 B8 BD (4)	D867 DE3D (4)	00029E3D (4)
葛	U+845B U+E0100	E8 91 9B E0 B8 80 (6)	845B E0100 (4)	0000845B 000E0100 (8)
𩸽	U+29E3D U+E0100	F0 A9 B8 BD E0 B8 80 (7)	D867 DE3D E0100 (6)	00029E3D 000E0100 (8)

- **A (U+0041):** ASCII文字である「A」は、UTF-8では`41`の1バイト、UTF-16では`0041`の2バイト、UTF-32では`00000041`の4バイトで表現される。UTF-8の効率の良さがわかる。

- **日 (U+65E5):** 「日」のような基本的な漢字（BMP内の文字）は、UTF-8では`E6 97 A5`の3バイト、UTF-16では`65E5`の2バイト、UTF-32では`000065E5`の4バイトで表現される。UTF-16では多くの漢字が2バイトで表現できるため、日本語のテキストデータにはよく用いられる。
- **鰹 (U+29E3D):** 「鰹」（ほっけ）のような追加面の文字は、UTF-8では`F0 A9 B8 BD`の4バイト、UTF-16ではサロゲートペアを用いて`D867 DE3D`の4バイト、UTF-32では`00029E3D`の4バイトで表現される。ここで、**UTF-16がサロゲートペアにより可変長**になっていることがわかる。
- **葛 (U+845B U+E0100):** 「葛」のようなIVS文字は、基底文字「葛」(U+845B) と異体字セレクタ(U+E0100)の組み合わせで表現される。結果、UTF-8では`E8 91 9B F3 A0 84 80`の7バイト、UTF-16では`845B DB40 DD00`の6バイト、UTF-32では`0000845B 000E0100`の8バイトとなる。ここで、**UTF-8とUTF-16がIVSによって可変長**になっていることがわかる。
- **鰹 (U+29E3D U+E0100):** 「鰹」のように、サロゲートペアで表現される文字にさらにIVSが付加される場合もある。この場合、UTF-8では`F0 A9 B8 BD F3 A0 84 80`の8バイト、UTF-16では`D867 DE3D DB40 DD00`の8バイト、UTF-32では`00029E3D 000E0100`の8バイトとなる。UTF-16ではサロゲートペアとIVSの組み合わせにより、さらに複雑な可変長表現となる。

可変長符号化方式の長所と短所

- **UTF-8:**
  - **長所:** ASCII文字との互換性が高く、英語圏のテキストデータではデータサイズを小さくできる。インターネット上で広く使われているため、互換性が高い。
  - **短所:** 日本語など、多くの漢字を含むテキストデータでは、UTF-16よりもデータサイズが大きくなることがある。また、文字によってバイト数が異なるため、文字単位の処理が複雑になる場合がある。

- **UTF-16:**
  - **長所:** BMP内の文字はすべて2バイトで表現されるため、日本語などを含むテキストデータでは、UTF-8よりもデータサイズが小さくなることがある。WindowsやJavaなどで内部的に使用されているため、これらの環境との親和性が高い。
  - **短所:** サロゲートペアやIVSを使用すると、文字によってバイト数が異なるため、文字単位の処理が複雑になる場合がある。ASCII文字も2バイトで表現するため、英語圏のテキストデータではデータサイズが大きくなる。

- **UTF-32:**
  - **長所:** IVSを除くすべての文字を4バイトで表現するため、文字単位の処理がシンプルになる。
  - **短所:** 他のエンコーディング方式と比べて、データサイズが大きくなる。

このように、サロゲートペアやIVSを用いると、特にUTF-16において、文字の表現が可変長になる。それぞれのエンコーディング方式には長所と短所があるため、用途に応じて適切な方式を選択することが重要である。

既存システムにおけるUnicode対応の難しさ

最後に、既存システムにおけるUnicode対応の難しさについて触れておく。汎用機で広く使われてきたCOBOLのような言語では、多くの場合、文字を固定長のデータとして扱うように設計されている。これは、当時のメモリが希少だった点や性能確保の問題が大きかったためである。

このような環境では、UTF-8のような可変長エンコーディングを扱うことが難しい。また、サロゲートペアやIVSを正しく処理するためには、プログラムの大幅な改修が必要となる。ましてや、扱う文字の性質によってはデータベースの論理設計や物理設計から見直さなくてはならない。これらの改修には膨大なコストと時間がかかる上、既存のコード資産に影響が波及する可能性も高い。

これらの修正を限られた予算と期間で実現するのは容易ではないため、汎用機上のCOBOLで記述されたアプリケーションは、サロゲートペアやIVSを含む文字に対応することが困難な場合が多い。結果として、これらのシステムでは、レガシーな文字コード（例えば、EBCDICベースの日本語文字コード）を使い続けることを選択せざるを得ない場合や、最新の文字セットが必要なアプリケーションの構築が難しいといった状況に陥っているケースが散見される。

この問題は、単に文字コードの問題に留まらず、古い技術から新しい技術への移行の難しさを示す典型的な例と言えるだろう。特に、Unicode化は基幹システムの場合、アプリケーション・プログラムだけでは無く、データベース、業務ロジック、運用手順への影響が多方面に発生する。そのため、Unicodeに対応することは、単に文字コードを変更するだけでなく、システム全体の見直しとモダナイゼーション（最新化）に繋がる大きな課題と言えるだろう。

Unicodeにおける絵文字の扱い

絵文字の登場と普及

近年、携帯電話やスマートフォンなどの情報機器で、絵文字が広く用いられるようになった。絵文字は、感情や事物を視覚的に表現するための記号であり、テキストベースのコミュニケーションを豊かにする役割を果たしている。

絵文字は、元々、日本の携帯電話キャリアが独自に開発したものである。1999年にNTTドコモがiモードサービスで絵文字を導入したのを皮切りに、KDDI（当時のDDIセルラーグループおよびIDO）やJ-フォン（現ソフトバンク）も追随し、各社が独自の絵文字セットを開発した。これらの絵文字は、キャリアのサービス差別化の要素として用いられ、ユーザー間のコミュニケーションを活性化する上で大きな役割を果たした。

スマートフォンの登場と絵文字の国際化

しかし、**スマートフォンの登場と、それに伴うグローバルなプラットフォームの普及は、絵文字の利用環境に大きな変化をもたらした**。特に、iPhoneの登場は、絵文字の国際化を加速させる契機となった。2007年に初代iPhoneが発売された当初、日本国外向けのモデルでは絵文字機能が搭載されていなかった。これは、当時の絵文字が日本のキャリアに依存した技術であり、国際標準が存在しなかったためである。しかし、日本人ユーザーの強い要望を受けて、2008年のiPhone OS 2.2のアップデートで、日本向けモデルに絵文字が搭載された。その後、2009年のiPhone OS 3.0では、絵文字キーボードが世界中で利用可能となった。

このiPhoneの対応は、絵文字の国際化に向けた大きな一歩となった。同時に、異なるキャリアやプラットフォーム間での絵文字の互換性問題も顕在化した。当初、絵文字はキャリアごとに異なるコード体系で運用されていたため、異なるキャリア間では、絵文字が正しく表示されない、いわゆる「文字化け」の問題が発生していた。この問題を解決するために、**Unicodeによる絵文字の標準化**が進められることとなった。

Unicodeによる絵文字の標準化

Unicodeへの絵文字の収録は、GoogleとAppleが共同で提案したことに始まる。両社は、2007年からUnicode Consortiumに対して絵文字の標準化を働きかけ、2010年のUnicode 6.0で、ついに絵文字が正式にUnicode標準に取り込まれた。このUnicode 6.0では、722個の絵文字が定義された。

AppleやGoogleが果たした役割は、単に標準化を推進しただけでなく、相互運用性の確保にも注力した点にある。特にAppleは、iPhoneで絵文字を世界的に普及させた立役者であり、その過程で蓄積したノウハウをUnicode標準化に活かした。例えば、異なる文化圏のユーザーが違和感なく使えるような絵文字のデザインガイドラインの策定や、既存の日本のキャリア絵文字との互換性確保などが挙げられる。

絵文字の符号化方式

Unicodeでは、絵文字は他の文字と同様に、コードポイントが割り当てられている。絵文字のコードポイントは、主にU+1F000以降の範囲に配置されている。絵文字は、UTF-8、UTF-16、UTF-32などのエンコーディング方式で符号化される。

ゼロ幅接合子(ZWJ)と絵文字

複数の絵文字を組み合わせて、1つの絵文字として表示する技術として、**ゼロ幅接合子**（ZWJ：Zero Width Joiner）が用いられる。例えば、「👧」（女性）と「⚕」（杖）の間にZWJを挿入することで、「👩‍⚕️」（女性の医療従事者）という1つの絵文字として表示することができる。

肌色修飾子と絵文字

一部の絵文字には、肌の色を指定するための**肌色修飾子**（Skin Tone Modifier）を付加することができる。肌色修飾子は、U+1F3FBからU+1F3FFまでの範囲に定義されている。例えば、「👍」（サムズアップ）に肌色修飾子を付加することで、異なる肌の色のサムズアップを表現することができる。

```
👍 U+1F44D  👍🏻 U+1F44D U+1F3FB 👍🏼 U+1F44D U+1F3FC 👍🏽 U+1F44D U+1F3FD
👍🏾 U+1F44D U+1F3FE 👍🏿 U+1F44D U+1F3FF
```

絵文字の異体字セレクタ

絵文字にも異体字セレクタが用いられることがある。例えば、「☺」というスマイリーフェイスの絵文字には、視覚的な表現を変化させるために、U+FE0E(VS15)やU+FE0F(VS16)という異体字セレクタが定義されている。

```
☺ U+263A          : 標準のスマイリーフェイス
☺ U+263A VS15 : テキスト表示のスマイリーフェイス
☺ U+263A VS16 : 絵文字表示のスマイリーフェイス
```

国旗の表現

2文字のアルファベットからなる国コードに対して、U+1F1E6からU+1F1FFまでの範囲のコードポイントを割り当て、2文字の組み合わせで国旗を表現する。例えば、日本の国旗は「🇯」+「🇵」で表現される。

```
🇯 U+1F1EF 🇵 U+1F1F5 : 日本の国旗
```

絵文字のUnicode標準化後の課題

Unicodeに収録されたことで、異なるキャリアやプラットフォーム間での絵文字の互換性は大幅に向上した。しかし、一方で新たな課題も生じている。

絵文字の意味の多様性

その一つが、**絵文字の意味の多様性**である。絵文字は、文化やコンテキストによって解釈が異なる場合がある。例えば、日本では「OK」の意味で使われる👌の絵文字が、他の国では侮辱的な意味を持つことがある。このような意味の違いは、異文化コミュニケーションにおいて誤解を生む可能性がある。

絵文字の乱用とコミュニケーションの質の低下

また、**絵文字の乱用や過剰な使用は、コミュニケーションの質を低下させる可能性**も指摘されている。絵文字は、テキストメッセージに感情やニュアンスを加える効果的な手段であるが、その使用が過剰になると、メッセージの内容が不明瞭になったり、相手に不快感を与えたりする可能性がある。

提供するベンダーによるデザインの違い

さらに、**絵文字のデザインがプラットフォームやフォントによって異なる**ことも、課題の一つである。同じUnicodeの絵文字であっても、Apple、Google、Microsoft、Twitterなど、提供するベンダーによってデザインが異なる。これによって、送り手が意図した感情やニュアンスが、受け手に正確に伝わらない可能性がある。

Unicodeでは、絵文字のデザインに関するガイドラインを定めているが、最終的なデザインは各プラットフォームベンダーの裁量に委ねられている。そのため、異なるプラットフォーム間での絵文字の表示の違いは、今後も継続する課題と考えられる。

結合文字 (Combining Character)

結合文字とは、基底文字に付加して、アクセント記号付きの文字や、濁点・半濁点を伴う文字などを表現するための文字である。例えば、「が」は基底文字「か」と結合文字「゛」（濁点）の組み合わせで表現できる。また、「é」は基底文字「e」と結合文字「´」（アキュートアクセント）の組み合わせで表現できる。

結合文字を用いることで、アクセント記号付きのアルファベットや、タイ語、ベトナム語などの声調記号付きの文字など、多様な文字を表現することができる。

Unicodeでは、結合文字は、基底文字の後ろに続けて配置される。複数の結合文字を基底文字に付加することも可能である。

例：

- か + ゛ -> が
- e + ´ -> é
- ハ + ゜ -> パ

行政事務における留意点: 外国人の氏名など、アクセント記号付きのアルファベットを扱う場合には、結合文字を正しく処理できることを確認する必要がある。また、結合文字の順序が異なると、見た目が同じでも異なる文字として扱われる場合があるため、注意が必要である。例えば、「か」に濁点と半濁点の両方を結合する場合、「か゛゜」と「か゜゛」は結果としては同じ「が」となっても、結合文字の順序が違うため、異なる符号となる。

その他のUnicodeの注意点

Unicodeを扱う際には、サロゲートペア、IVS、結合文字以外にも、注意すべき点がいくつかある。

正規化形式 (Normalization Forms)

Unicodeの世界では、画面上で同じに見える文字でも、内部的な表現方法が複数存在する。これは、同じ料理を作るのに複数のレシピがあるようなものだ。例えば、「が」という文字。我々は一つの文字として認識するが、コンピュータ内部では「か」という文字と「゛」（濁点）という記号の組み合わせとして表現することも可能である。

複数の表現方法が存在する理由

これは、Unicodeが多様な言語や歴史的な文字を扱えるように設計された結果である。特に、古い文字コードとの互換性を維持するために、このような柔軟性が必要となった。

具体的な例: 「が」の表現

- **正規結合 (NFC: Normalization Form Canonical Composition)**
  - コンピュータは、「が」を **単一の文字** として扱う。
  - Unicodeの内部的な番号（コードポイント）は **U+304C** である。
  - Hex表現 (UTF-8) は一般的に `E3 81 8C` となる。（環境により異なる場合がある）
- **正規分解 (NFD: Normalization Form Canonical Decomposition)**
  - コンピュータは、「が」を **「か」と「゛」の組み合わせ** として扱う。
  - 「か」のコードポイントは **U+304B**、濁点のコードポイントは **U+3099** である。
  - Hex表現 (UTF-8) は一般的に `E3 81 8B` (か) と `E3 82 99` (゛) の2つに分かれる。

このように、見た目は同じ「が」でも、内部的な表現が異なる場合がある。

他の正規化形式

- **互換結合 (NFKC: Normalization Form Compatibility Composition)**

- これは、見た目は似ているが、意味や用途が異なる文字を、代表的な文字に **変換・統合** する処理を含む。さらに、NFCと同様に可能な限り単一の文字に合成する。
- 例：「①」（U+2460、丸付きの1）を半角数字の「1」（U+0031）に変換してから正規化する。
- Hex表現の例：`EF 91 A0` (①) が `31` (1) に変換される可能性がある。

- **互換分解 (NFKD: Normalization Form Compatibility Decomposition)**

  - これは、NFKCと同様に互換性のある文字を分解するが、最終的に結合せずに、結合文字を優先する形式である。
  - 例：「㈱」（U+3231、株式会社の記号）を「(」 (U+0028)、「株」 (U+682A)、「)」 (U+0029) に分解する。
  - Hex表現の例：`E3 88 B1` (㈱) が `28` (( )、`E6 A0 AA` (株)、`29` ( )) に分解される。

具体例：検索時の問題

「株式会社ABC」という名前を検索したいとする。

- あるシステムでは、「株」は `U+682A` で保存されている（NFC）。
- 別のシステムでは、「㈱」(`U+3231`) で保存されている（互換文字）。

この場合、単純に文字列を比較するだけでは、一致しないと判断されてしまう。

正規化の重要性：システム間連携を円滑に

異なるシステム間で文字データをやり取りする際、お互いが異なる正規化形式を使用していると、以下のような問題が発生する可能性がある。

- **検索の失敗:** 上記の例のように、同じ意味の言葉でも、表現が異なると検索で見つからない。
- **データの不一致:** データベースで照合を行う際に、見た目は同じデータが異なるものとして扱われる。
- **文字化け:** 特に互換文字を含む場合に、予期しない文字に変換されてしまう。

このような問題を避けるためには、システム間でデータをやり取りする前に、**どちらかの正規化形式に統一する** という処理が必要となる。例えば、すべてのデータをNFCに正規化することで、「が」は常に `U+304C` として扱われるようになり、システム間の認識のずれを防ぐことができる。

双方向テキスト (Bidirectional Text)

アラビア語やヘブライ語のように右から左に書く文字と、日本語や英語のように左から右に書く文字が混在する文書では、文字の表示順序を正しく制御する必要がある。Unicodeでは、このような双方向テキストを扱うための仕組みが用意されている。

代替テーブル (Substitution Table)

異なる文字セット間では、同じ文字であっても異なる符号が割り当てられている場合がある。そのため、ある文字セットで符号化されたデータを、別の文字セットで扱うためには、文字コードの変換が必要となる。

文字コードの変換を行うためには、文字セット間の対応関係を定めた**対応表**（または**変換テーブル**）が必要となる。対応表には、一方の文字セットの各文字について、対応する他方の文字セットの文字（または符号）が記述されている。

しかし、ある文字セットに含まれる文字が、別の文字セットに含まれていない場合、完全な対応表を作成することはできない。このような場合、代替の文字や記号を用いて、元の文字を近似的に表現することがある。例えば、JIS X 0208には含まれているが、Shift_JISには含まれていない文字をShift_JISで表現する場合、以下のような代替文字が用いられることがある。

- 「Ⅲ」（ローマ数字の3）→「III」（アルファベットのIを3つ並べる）
- 「①」（丸付き数字の1）→「(1)」（括弧で囲まれた数字の1）
- 「㍻」（元号の平成）→「平成」

このような代替文字の対応関係を定めた表を、**代替テーブル**と呼ぶ。代替テーブルは、文字コード変換において、完全な対応が不可能な場合に、次善の策として用いられる。

東アジアの漢字文字コード事情

東アジア地域は、悠久の歴史の中で漢字を共通の文字基盤として発展してきた。しかし、それぞれの国や地域において、言語や文化、そして行政上の必要性から、漢字の字形や使用頻度に差異が生じ、それらを情報システム上で扱うための文字コードも独自に発展してきたのである。文字コードは単に文字をデジタルデータとして表現する技術的な仕様にとどまらず、各地域の文化的なアイデンティティや行政システムに深く根ざした側面を持つ。

現在、広く利用されているOpenTypeフォントの技術仕様では、1つのフォントファイルに含めることができるグリフ（文字の図形）の数は最大で65,536個に制限されている。これは、初期のコンピュータ技術における設計上の制約によるものだ。しかしながら、後述するように、各国で行政が利用する文字セットは肥大化の一途を辿っており、このOpenTypeフォントのグリフ数制限が、今後の情報処理やフォント技術の発展に大きな影響を与えることが懸念されるのである。本稿では、中国、台湾、韓国、北朝鮮における漢字文字コードの現状を概観し、特に行政向け文字セットの肥大化がOpenTypeフォントの仕様に与える影響について考察する。

各国・地域における漢字の文字数

地域	一般向け	文字数	行政向け	文字数
日本	JIS X 0213:2004	11,233	行政事務標準文字	69,839
中国	GB 18030 Level 2	27,780	GB 18030 Level 3	87,887
台湾	Big5	13,060	CNS 11643	108,629
香港	Big5-HKSCS	17,323	増補HKSCS-2016	22,356

中国

中国の文字コードは、GB（Guobiao、国家標準）によって標準化されている。一般向けの文字コードとしては、GB 2312、GBK、そしてGB 18030が挙げられる。GB 18030は、中国国内で販売される全ての情報技術製品に実装が義務付けられている重要な文字コードだ。Level 2には27,780字の漢字が含まれており、日常的な使用には十分な文字数をカバーする。

さらに、行政や専門分野での利用を想定したGB 18030 Level 3には、87,887字もの漢字が収録されている。これには、古籍に使われるような漢字や、少数民族の言語で使用される文字なども含まれている。中国における文字コードの標準化は、情報化社会の発展を支える上で重要な役割

を果たしており、GB 18030は、その最新の成果と言えるだろう。しかしながら、その一方で、収録文字数の多さは、フォント開発やソフトウェアの互換性において、新たな課題も提起しているのである。

台湾

台湾では、Big5という文字コードが伝統的に使用されてきた。Big5には13,060字の繁体字が含まれており、台湾の日常的な情報処理において広く普及している。しかし、Big5は商業的な成功を収めたものの、その拡張性には限界があり、収録されていない漢字も存在した。そのため、より多くの漢字を扱う必要のある行政分野などでは、CNS 11643という文字コードが用いられている。

CNS 11643は、中華民国国家標準（Chinese National Standard）として制定された文字コードであり、その最大の特徴は、108,629字という非常に多くの漢字を収録している点にある。これは世界最大級の文字コードの一つであり、台湾の複雑な歴史的背景と文字文化への強い意識を反映した結果と言えるだろう。CNS 11643の策定と運用には、政府機関だけでなく、学術機関や業界団体も関わっており、新しい文字の追加には、提案、審査、承認といった厳格な手続きが設けられている。文字の追加には費用が発生する場合もあり、その字形や使用例などの詳細な情報提供が求められる。

CNS 11643の文字数が増加してきた背景には、古典籍のデジタル化、人名や地名などの固有名詞への対応、そして異体字の網羅的な収録といった要因が挙げられる。特に、台湾においては、伝統的な文字文化の保存に対する意識が高く、それがCNS 11643の巨大化を後押ししてきたと言えるだろう。しかし、CNS 11643の膨大な文字数は、フォントの作成や管理を非常に困難にしている。OpenTypeフォントの制限下では、CNS 11643の全ての文字を一つのフォントファイルに収めることは不可能であり、複数のフォントを組み合わせるなどの複雑な対応が必要となる。また、Unicodeへの移行も進められているが、CNS 11643の持つ独自性から、完全な移行には至っていないのである。

香港

香港では1980年代からMS-DOSの普及に伴い、台湾のソフトウェアやハードウェアを使用していたため、台湾で制定されたBig5がデファクトスタンダードとして普及していた。しかし、Big5は台湾の標準中国語を表記するための漢字を収録したものであり、香港の広東語に特有の方言字は十分に収録されていなかった。このため、香港固有の文字を表現する際には外字として個別に対応せざるを得ない状況が続いていた。

1984年から1994年までの期間、香港では素のBig5（13,060字）を使用していたが、地名や人名、広東語特有の文字については外字として個別対応する必要があった。この状況を改善するため、1995年に香港政庁は政府機関間の電子通信を円滑にする目的で、政府通用字庫（GCCS）を制定し、3,049文字を追加収録した。

その後、1999年にGCCSは香港増補字符集（HKSCS）として改定され、4,702文字へと拡充された。HKSCSはその後も継続的に改定が行われ、2001年に4,818文字、2004年に4,941文字、2008年に5,009文字、2016年に5,033文字へと段階的に拡充されてきた。

現在のHKSCSには、香港の地名、人名用漢字、広東語用文字、異体字、一部の簡体字、ひらがな、カタカナ、キリル文字など、香港で一般に使用される文字が含まれている。香港では、Big5をベースとしつつHKSCSを組み合わせた文字セット（Big5-HKSCS）が標準的に使用されており、これにより香港固有の文字表現の課題は大きく改善されている。

韓国

韓国の文字コード体系

韓国では、かつて KS X 1001 (旧称 KS C 5601) が基本的な文字コード規格として使用されていた。これは、ハングル 2,350 字、漢字 4,888 字、その他英数字や日本語の仮名文字など 989 字の合計 8,227 字を収録していた。これを補完する形で KS X 1002 が存在し、古ハングル音節文字や字母、補助漢字、記号などが追加収録されていた。しかし、現在では Unicode が主要な文字コードとして広く使用されており、KS X ISO/IEC 10646 として規格化されている。

文字コードの特徴と変遷

KS X 1001のハングル部分は「完成型コード」(組み合わせ型ではない) を採用しており、あらかじめ組み合わされたハングル音節のみを収録していた。そのため、現代韓国語で使われるすべての可能なハングル音節の組み合わせ (11,172 字) を網羅しているわけではなく、外来語に使用される音など、表現できない音節が存在するという課題があった。この問題に対処するため、1992 年に KS X 1001は改定され、現代ハングル 11,172 字全てを収録できるように拡張された。しかし、この改訂版はあまり普及しなかった。

その後、1996 年の Unicode 2.0 で 11,172 文字すべての現代ハングル文字が収録され、以降Unicodeが韓国語の文字コードとして広く普及するようになった。 Unicodeは、現代ハングルだけでなく、古ハングル、漢字、その他世界中の多くの文字を収録している。

韓国の戸籍に当たる家族関係登録簿に名前として登録できる漢字は、大法院（日本における最高裁判所に相当）が人名用漢字のリストを発行しており、2015年時点で8,142文字が収録されている。しかしながら全てがUnicodeに収録されている訳ではなく、外字としてしか表現できない文字も残存している。

인명용 한자표

(호적법시행규칙 제37조)

한글	한문 교육용 기초한자 (2000. 12. 31. 현재)	인명용 추가 한자						
		(1991. 4. 1.)	(1994. 9. 1.)	(1998. 1. 1.)	(2001. 1. 4.)	(2003. 10. 20.)	(2005. 1. 1.)	(2007. 2. 15.)
가	家佳街可歌加價假架暇	嘉嫁稼賈駕	伽	迦柯	呵哥枷珂痂苛袈訶跏軻茄			笳
각	各角脚閣却覺刻	珏恪殼			慤			
간	干間看刊肝幹簡姦懇	艮侃杆玕竿揀諫墾		栞	奸柬桿澗癎磵稈艱			
갈	渴	葛			羯喝曷碣竭褐蝎鞨			
감	甘減感敢監鑑(�鑒)	勘堪瞰			坎嵌憾戡柑橄疳紺邯龕			
갑	甲	鉀			匣岬胛閘			
강	江降講强(強)康剛鋼(鋼)綱	杠堈岡崗姜橿彊慷			畺疆糠絳羌腔舡薑襁鏘		嬢跫	
개	改皆個(箇)開介慨槪蓋(盖)	价凱塏漑			塏愾疥芥豈鎧		玠	

図1：韓国大法院の発行する人名用漢字リスト

韓国における人名用漢字制限の合憲判決

韓国では、子の出生を届け出る際、その名前に使用できる漢字が「通常使われる漢字」（人名用漢字）に制限されている。これは、「家族関係の登録等に関する法律」第44条第3項および同規則第37条（以下、併せて「審判対象条項」）に基づくものであり、人名用漢字以外の漢字を用いた場合、出生届は受理されるものの、戸籍にあたる家族関係登録簿にはハングルのみで記載されることとなっている。

2016年7月28日、韓国憲法裁判所は、この人名用漢字制限の合憲性について判断を下した（2015헌마964）。この事件の発端は、ある夫婦が、出生した子の名前に「嫘」を含む漢字を使用しようとしたところ、この漢字が人名用漢字に含まれていなかったため、家族関係登録簿にハングルのみで子の名前が記載されたことにある。この夫婦は、審判対象条項が、憲法で保障された「両親が子の名を付ける自由」を侵害し、人格権および幸福追求権を侵害しているとして、憲法訴願を提起した。

憲法裁判所は、裁判官9名の合議体（合憲6名、違憲3名）で審判対象条項を**合憲**と判断した。多数意見は、まず、名の重要性に鑑み、国家は人名に対して一定の規律を加えることができるとし、本件条項は、通常使われない難しい漢字の使用による当事者や利害関係者の不便を解消し、行政の電算化に対応するという正当な目的があり、その手段も適切であると判示した。

さらに、侵害の最小性および法益の均衡性についても検討を加えた。人名用漢字は現在8,142字であり、日本の人名用漢字や中国の「通用規範漢字表」と比較しても極端に少ないとは言えないこと、人名用漢字は継続的に追加されており、不便解消の補完装置が作動していることを指摘した。また、人名用漢字以外の漢字を使用しても出生届自体は受理され、家族関係登録簿にはハングルで記載されること、私的生活で当該漢字名の使用が妨げられるわけではないこと、一定の例外規定も存在することから、過剰な制限とは言えず、公益との法益バランスも維持されていると判断した。

これに対し、3名の裁判官は反対意見を述べた。反対意見は、名は人格の象徴であり、「両親が子の名を付ける自由」は憲法上保障されるとした上で、韓国の社会・文化的背景を考慮すると、子の名に使える漢字の制限には慎重な検討が必要であると主張した。また、審判対象条項の立法目的は現代社会においては正当性を失っており、漢字の使用制限は侵害の最小性を満たしておらず、法益の均衡性も欠いていると批判し、違憲であると結論付けた。

2名の裁判官による補足意見は、多数意見に賛同しつつ、「両親が子の名を付ける自由」は子の氏名権と人格権を侵害しない範囲で行使されなければならないと指摘した。その上で、難しい漢字を含む名の登録制限は、本人の社会生活に支障を与えかねないため、制限には合理性があり、子の福利や権益を侵害するものではないと付言した。

最終的に、憲法裁判所は、審判対象条項が過剰禁止原則に違反せず、請求人の「子の名を付ける自由」を侵害するものではないと判断し、**合憲**と結論付けた。

現代韓国における漢字使用の状況

現代の韓国語では、1948年の「ハングル専用法」制定以降、公文書や一般的なコミュニケーションにおいて漢字の使用が極めて限定的となっている。日常的な漢字混用は以下の場面に限られている：

- 仏教関連書籍
- 法学書籍
- 外国人観光客向けの道路標識の地名表記

漢字教育の現状と展望

2014年に朴槿恵政権が漢字教育復活を試みたものの、ハングル関連団体からの反発により、漢字併記から脚注表記に変更された。2020年12月には、初等学校の教科書への漢字併記を可能とする法改正案が提出され、再び漢字論争が起きている。

注目すべきは、現代韓国語の語彙の約70%が漢字語であり、専門用語においては90%にまで及ぶという点である。つまり、文字表記としての漢字は使用されていないものの、語彙体系における漢字の影響は依然として強く残っている。

北朝鮮

北朝鮮の文字コードKPS 9566は、1993年の初版から複数回の改訂を経て、現在はKPS 9566-2011（一部では2012とも呼ばれる）が最新規格となっている。これに加えて、2000年に制定されたKPS 10721は19,469字以上の漢字を収録しているとされる。

北朝鮮内での実際の運用においては、KPS 9566の処理系はほとんど存在せず、韓国のEUC-KRを使用することが一般的であった。ただし、北朝鮮製Linuxディストリビューション「Red Star OS」では、KPS規格が採用されている。

最新規格であるKPS 9566-2011は126×178の文字集合で構成され、以下の特徴を持つ。特殊文字領域には符号、絵文字、チョソングルの字母、平仮名、片仮名、ギリシア文字、キリル文字などが含まれる。チョソングル領域には2,679字が北朝鮮の字母配列に従って配列され、漢字領域には4,653字がチョソングルの読音順に配列されている。

特筆すべき特徴として、朝鮮労働党のハンマー、鎌、筆の紋章や、指導者の名前に関する特別な実装がある。金日成、金正日の名前を表す6文字に加え、金正恩政権後に新たに3文字が追加され、現在は計9文字が存在している。そのため技術面では、Unicodeとの互換性において一部の課題が存在する。特に特殊記号や指導者の名前については、Unicodeに含まれていないため、完全なラウンドトリップ（往復変換）が実現されていない。Microsoft Windowsはライセンス上、北朝鮮への輸出を禁じているため、WindowsにおいてKPS 9566を公式に実装した製品は存在していない。

ベトナムにおける漢字の取り扱い

東アジアにおける漢字情報処理において、しばしば言及されるベトナムも、かつては漢字文化圏に属していた。独自の文字であるチュノム（Chữ Nôm）は、漢字を基にベトナム語を表記するために発展した文字であり、豊富な数の文字が存在した。しかし、現代ベトナムでは、クオック・グー（Quốc Ngữ）と呼ばれるラテンアルファベット表記が公用文や日常 ব্যবহার の中心となっている。それでも、歴史的な文献や一部の専門分野においては漢字やチュノムが用いられる場面があり、これらの文字のデジタル化や情報処理も課題となっている。

東アジアの漢字文字コードにおける課題

行政向け文字セット肥大化の影響

上記のように、東アジア各国では、それぞれの歴史や文化、行政上の必要性から、独自の漢字文字コードが発展してきた。特に行政分野で使用される文字セットは、人名や地名など、正確な記録を保持するために、非常に多くの漢字を含む傾向にある。しかしながら、OpenTypeフォントの仕様では、1つのフォントファイルに含めることができるグリフの数が65,536個に制限されている。

このOpenTypeフォントのグリフ数制限は、特に文字数の多い行政向け文字セットを扱う上で大きな課題となる。例えば、台湾のCNS 11643のように、10万字を超える文字セットを単一のOpenTypeフォントファイルで実現することは不可能である。このため、これらの文字をデジタル環境で利用するためには、複数のフォントファイルを連携させたり、高度なフォント技術を利用したりするなどの複雑な対応が必要となるのである。もっとも、OpenTypeの仕様は固定されたものではなく、将来的な拡張の可能性も議論されている。技術的な進歩や、Unicodeの更なる普及、そして多様な文字への需要の高まりによっては、OpenTypeのグリフ数制限が緩和または撤廃される未来も考えられる。

今後のフォント技術の発展においては、OpenTypeフォントのグリフ数制限を克服する技術や、より効率的なフォント管理方法の開発が求められる。例えば、可変フォント技術を活用することで、必要な字形のみを生成し、フォントファイルのサイズを抑えることが可能になるかもしれない。また、クラウドフォントなどの技術を利用することで、ユーザーの環境にフォントをインストールすることなく、必要な文字を都度配信するといった仕組みも考えられる。

漢字文字コードの国際相互運用へ向けた展望

東アジア各国における漢字文字コードの現状を見ると、それぞれの国や地域が持つ歴史的、文化的背景、そして行政上の必要性によって、多種多様な文字コードが発展してきたことがわかる。特に行政分野で使用される文字セットは、正確性を期すために肥大化の傾向にあり、OpenTypeフォントのグリフ数制限という技術的な制約との間で、大きな課題が生じている。

この課題を解決するためには、フォント技術の革新や、文字コードの標準化に向けた国際的な協力が不可欠である。グローバルな情報化社会において、円滑な情報流通を実現するためには、文字コードの相互運用性を高め、異なる環境間での文字化けを防ぐための取り組みが重要となる。今後の技術開発と国際協力によって、東アジアにおける漢字情報処理の未来がより良い方向へ進むことが期待される。

文字コードの実装

主要OSと文字コード

オペレーティングシステム（OS）は、コンピュータ上で動作するソフトウェアの基盤であり、文字コードの扱いにも大きな影響を与える。ここでは、Windows、macOS、iOS、Android、ChromeOSという主要なOSにおける文字コードの扱いについて説明する。

Windowsの文字コード

現代のWindowsでは、内部的にはUnicode（UTF-16）が用いられているが、外部とのインターフェースでは地域別の文字コード（コードページ）が用いられる。

日本語版DOSの文字コード

コードページとは、Windowsで用いられる文字コード体系のことである。各コードページには、特定の言語や地域で用いられる文字が収録されている。例えば、日本語のコードページには、Shift_JISをベースとしたコードページ932がある。

日本語版DOSでは、**コードページ932**が用いられていた。コードページ932は、Shift_JISをベースとしているが、NEC特殊文字やIBM拡張漢字などの、一部の機種依存文字が追加されている。

Windows 3.0以前の日本語対応

Windows 3.0以前の時代、日本語対応のMS-DOS/MS-Windowsは、メーカーごとに異なる製品として提供されていた。さらに、漢字の字形規格がJIS C 6226-1978(いわゆるJIS78)かJIS C 6226-1983(いわゆるJIS83)か、搭載されているフォントも各社バラバラであった。

この結果、アプリケーションの互換性問題が発生したり、ユーザーが異なる環境に慣れる必要が生じたりするなど、現在とは異なる不便さが存在した。

日本語版Windowsの文字コード

日本語版 Windowsでは、長い期間にわたって **シフト JIS** をベースとした **コードページ 932** (CP932) が、標準的な文字エンコーディングとして用いられてきた。しかし、Windows NT系では内部処理に **Unicode** を採用し、現在に至るまでUnicodeベースのシステムへと移行している。

以下、各OSリリースの詳細と、文字コードに関する変遷について述べる。

Windows 3.1 (1992年)

このバージョンは、MS-DOS上で動作する16ビットのオペレーティング環境である。ファイル名やテキストファイルのエンコーディングとしては、シフト JIS (JIS X 0208-1990準拠) に基づくコードページ932が使用されていた。内部処理も同様にコードページ932ベースで行われていた。この時点ではUnicodeはサポートされていない。

Windows 3.1の特筆すべき点として、TrueTypeフォントへの対応が挙げられる。それまでのWindowsでは、ビットマップフォントが主に使われていたが、Windows 3.1では、スケーラブルなアウトラインフォントであるTrueTypeが採用された。これにより、画面表示や印刷において、滑らかで美しい文字表現が可能になった。

また、日本語TrueTypeフォントとして、MS 明朝とMS ゴシックが標準搭載されたことも重要な変化である。これらのフォントは、それまで各メーカーが独自に提供していた日本語フォントに代わり、Windowsの標準的な日本語フォントとして広く普及した。

さらに、Windows 3.1の登場により、Windowsというプラットフォーム上では、どのメーカーのPCでも表示される字形(主にJIS83字形)と外字セットが統一されたことも大きな意義を持つ。それまで、日本では各メーカーが主にシフトJISを採用してはいたものの、JIS78字形とJIS83字形の違いや、各社独自の外字を使用していたため、異なるシステム間でのデータ交換に問題が生じることが多かった。

Windows 3.1で、標準フォントと外字セット(およびその配置されたコードポイント)が統一されたことは、日本語の情報処理環境の発展に大きく貢献したと言える。ただし、完全な文字セット(文字コード)の統一は、Windows NT系でUnicodeが採用されるまで待たなければならなかった。

Windows 95/98/Me (1995年〜2000年)

これらのバージョンは、MS-DOSとWindowsが統合されたOSである。引き続き、ファイル名やテキストファイルのエンコーディングとしては、**シフトJIS (JIS X 0208-1990準拠)** ベースの**コードページ932**が使用されていた。内部処理についてもコードページ932ベースで行われており、Unicodeはサポートされていなかった。

Windows 9x系がUnicodeに本格対応しなかった理由として、当時のハードウェア環境、特にメモリ容量の制約が大きかったことが挙げられる。Unicode (UTF-16) では、多くの文字を2バイトで表現するため、シフトJISと比較してメモリ使用量が増加する。当時のPCのメモリ容量は限られており（例えば、Windows 95の推奨メモリ容量は8MB）、Unicodeの採用によるメモリフットプリントの増大は、システム全体のパフォーマンスに大きな影響を与える可能性があった。

加えて、当時のアプリケーションの多くがシフトJISベースで開発されていたことも、Unicode対応を難しくした要因の一つである。Unicodeに対応するためには、アプリケーションの大幅な改修が必要となり、開発者にとって大きな負担となっていた。

これらの理由から、Windows 9x系では、パフォーマンスと互換性を重視し、Unicodeの本格的なサポートは見送られた。ただし、限定的なUnicode APIは存在していたため、一部のアプリケーションではUnicodeを使用することが可能だった。

Windows NT 3.1/3.5/3.51 (1993年〜1995年)

初期のNT系は、ビジネス用途をターゲットとした32ビットOSである。**内部処理に初めてUnicode (UTF-16, Unicode 1.1) が採用された**。ただし、当時のUnicodeは、まだ補助漢字面(いわゆるサロゲートペア)をサポートしておらず、扱える文字は基本多言語面 (BMP) に限定されていた。ファイル名としてはUnicodeが使えたが、APIは限定的で、テキストファイルのエンコーディングとしては、引き続き**シフトJIS (JIS X 0208-1990準拠)** ベースの**コードページ932**が使用されていた。

Windows NT 4.0 (1996年)

このバージョンでは、内部文字コードは引き続き**Unicode 1.1 (UTF-16)** が用いられた。**後述のXKP**による標準外字セットが実装されたが、当時のUnicodeは、補助漢字面をサポートしておらず、サロゲートペアには非対応であった。ファイル名やテキストファイルのエンコーディングとしては、引き続き**シフトJIS (JIS X 0208-1990準拠)** ベースの**コードページ932**が使用されていた。

Windows 2000 (2000年)

Windows 2000では、内部文字コードが **Unicode 2.0 (UTF-16)** にアップデートされた。補助漢字面の一部が追加され、限定的にサロゲートペアに対応した。API も拡充され、本格的にUnicodeベースのアプリケーション開発が可能になった。テキストファイルのエンコーディングとしては、引き続き**シフト JIS (JIS X 0208-1990, JIS X 0212-1990 準拠)** ベースの**コードページ 932** がデフォルトで使用されていたが、Unicodeでエンコードされたテキストファイル（UTF-16, UTF-8）を扱うことも可能になった。

Windows XP (2001年)

Windows XPでは、内部文字コードは引き続き **Unicode 2.0 (UTF-16)** が用いられたが、サロゲートペアに完全対応した。これにより、補助漢字面を含む多くの漢字を扱うことが可能になった。また、**JIS X 0212-1990** 補助漢字にも対応した。 テキストファイルのエンコーディングとしては、**シフト JIS (JIS X 0208-1990, JIS X 0212-1990 準拠)** ベースの**コードページ 932** がデフォルトで使用されていた。

Windows Vista (2006年)

Windows Vistaでは、内部文字コードが **Unicode 3.2 (UTF-16)** にアップデートされ、JIS X 0213:2004 に対応した。これにより、いわゆる「**JIS第3水準漢字**」「**JIS第4水準漢字**」を含む多くの漢字が扱えるようになった。テキストファイルのエンコーディングとしては、引き続き**シフト JIS** ベースの**コードページ 932** がデフォルトで使用されていたが、Unicodeでエンコードされたテキストファイルの使用がより一般的になった。

Windows 7 (2009年)

Windows 7では、内部文字コードが **Unicode 5.0 (UTF-16)** にアップデートされた。IVS (異体字シーケンス) に限定的に対応し、一部の異体字を表現できるようになった。引き続き **JIS X 0213:2004** に対応。テキストファイルのエンコーディングとしては、**シフト JIS** ベースの**コードページ 932** がデフォルトで使用されていた。

Windows 8 (2012年), Windows 8.1 (2013年)

Windows 8/8.1では、内部文字コードが **Unicode 6.0 (UTF-16)** にアップデートされ、IVSに完全対応した。これにより、IVSを用いて定義された異体字を正しく表示・入力できるようになった。引き続き **JIS X 0213:2004** に対応。テキストファイルのエンコーディングとしては、**シフト JIS** ベースの**コードページ 932** がデフォルトで使用されていた。

Windows 10 (2015年)

Windows 10では、内部文字コードが **Unicode 9.0 (UTF-16)** にアップデートされた。引き続き **JIS X 0213:2004** に、IVSにも完全対応している。テキストファイルのデフォルトエンコーディングは、**シフトJIS**ベースの**コードページ932**であるが、UTF-8の利用が以前のバージョンよりも容易になっている。バージョン1903以降、UTF-8をデフォルトのコードページとして設定可能になった (ただし、非推奨)。

Windows 11 (2021年)

Windows 11では、内部文字コードが **Unicode 15.0 (UTF-16)** にアップデートされた。引き続き **JIS X 0213:2004** に、IVSにも完全対応している。依然としてテキストファイルのデフォルトエンコーディングは、**シフトJIS**ベースの**コードページ932**であるが、UTF-8の使用がより一般的になりつつある。

日本語版Windowsの文字コード対応年表

日本語版 Windowsでは、Windows NTおよび Windows 2000以降では内部的には Unicode（UTF-16）が用いられているが、外部とのインターフェースでは、コードページ932が用いられる。

OSバージョン	文字コード	SP	IVS	JIS規格
Windows 3.1	Shift_JIS	非対応	非対応	0208-1990
Windows 95/98/Me	Shift_JIS	非対応	非対応	0208-1990
Windows NT 3.x	Unicode 1.1	非対応	非対応	0208-1990
Windows NT 4.0	Unicode 1.1	非対応	非対応	0208-1990
Windows 2000	Unicode 2.0	限定	非対応	0212-1990
Windows XP	Unicode 2.0	対応	非対応	0212-1990
Windows Vista	Unicode 3.2	対応	非対応	0213:2004
Windows 7	Unicode 5.0	対応	限定	0213:2004
Windows 8/8.1	Unicode 6.0	対応	対応	0213:2004
Windows 10	Unicode 9.0	対応	対応	0213:2004
Windows 11	Unicode 15.0	対応	対応	0213:2004

Windows Vistaの字形変更による混乱

JIS2004の字形変更は、それまで広く使われていた日本語環境に大きな影響を及ぼした。特に、2007年に発売された**Windows Vista**の登場は、この影響を一般ユーザーにまで広く認知させる契機となった。

Windows Vistaでは、標準の日本語フォントとして、JIS2004に準拠した「**メイリオ**」や「**MS ゴシック/明朝 ver.5.0**」が採用された。これによって、Windows Vista搭載のPCでは、JIS2004で変更された字形が標準的に表示されるようになった。

しかし、この変更は大きな混乱を引き起こした。Windows XP以前で使われていたJIS90字形に慣れ親しんでいたユーザーにとって、JIS2004で変更された字形は、別の文字に見えたり、違和感を覚えたりすることが多かった。特に、人名や地名などの固有名詞で字形が変わると、文書の意味が正しく伝わらない、あるいは誤解を生む可能性があった。

具体的な混乱の例としては、以下のようなものが挙げられる。

- **「葛」の字形変更問題:**

  「葛」の字は、JIS90（JIS X 0208-1990）では、下部が「ヒ」の部分で構成される字体「葛」で表示されていた。しかし、JIS2004（JIS X 0213:2004）の改正により、下部が「人」となる字体「葛」に変更された。

  この変更により、新しいOS（Windows Vista以降）やフォントでは「葛」が標準となり、以前の「ヒ」の字体「葛」を表示することが難しくなった。葛飾区では本来の字形が表示できるようになった一方で、葛城市では市制施行時（平成16年）にパソコンでの利便性を考慮して葛を採用していたため、JIS2004対応OS（Windows Vista以降）では意図した字体と異なる表示になるという問題が発生した。
  なお、現在では両方の字体が正式な表記として認められており、官公署では字体の違いを理由に申請や届出を不受理とすることはない。

- **一点しんにょうと二点しんにょうの問題:**

  JIS 2000までは「辻」「辺」などの一点しんにょうだった漢字が、JIS2004からは二点しんにょう「辻」「辺」に変更された。この変更により、人名などで表記が変わり、混乱が生じた。

これらの字形変更は、企業や自治体などにおいても大きな問題となった。特に、住民基本台帳などの公的文書を扱うシステムでは、JIS2004への対応が急務となった。しかし、システムの改修には多大なコストと時間がかかるため、一部の自治体や官公庁では当面の間JIS90互換フォントを入れて整合を取るなど、対応が遅れるケースも少なくなかった。

字形選択の必要性とIVSの実装

Windows Vistaの経験は、OSレベルで複数の字形を使い分ける必要性を強く認識させるものだった。特に、固有名詞などにおいては、環境によらず正しい字形で表示することが重要である。この問題を解決する手段として注目されたのが、**IVS (Ideographic Variation Sequence)**である。

IVSは、Unicodeで規定されている異体字セレクタを用いて、基底文字に続く異体字セレクタによって、その文字の異体字を指定する仕組みである。例えば、「葛」の基底文字に、特定の異体字セレクタを付加することで、JIS2000字形とJIS2004字形のどちらを表示するかを指定することができる。

Windows 7では、このIVSが実装され、OSレベルで異体字の使い分けが可能になった。これにより、アプリケーションやユーザーは、必要に応じて、特定の字形を選択して表示・入力することができるようになった。例えば、人名や地名などの固有名詞を登録する際に、IVSを用いて正しい字形を指定することで、環境によらず正確な表記を実現することが可能となった。

Windows 7におけるIVSの実装は、JIS2004の混乱を収束させ、日本語環境における文字表現の正確性を向上させる上で、重要な役割を果たしたと言える。しかし一方で、IVSを利用するためには、アプリケーション側の対応も必要であり、完全に普及するにはまだ課題が残されている。

Windows APIと文字コード

Windows APIには、ANSI版とUnicode版の2種類がある。ANSI版のAPIは、文字列をコードページで扱う。一方、Unicode版のAPIは、文字列をUTF-16で扱う。

文字列処理関数の使い分け

ANSI版のAPIとUnicode版のAPIでは、文字列を扱う関数が異なる。例えば、文字列の長さを取得する関数は、ANSI版では`strlen`であるが、Unicode版では`wcslen`である。

.NET Frameworkの文字コード

.NET Frameworkでは、内部的にはUnicode（UTF-16）が用いられている。また、各種エンコーディングへの対応も提供されている。

macOSの文字コード

macOSでは、内部的にはUnicode（UTF-8）が用いられている。

漢字TalkからmacOS (Classic Mac OS)

初期の日本語対応のMacintosh OS（漢字Talk）では、**MacJapanese**という文字コードが用いられていた。MacJapaneseは、Shift_JISをベースとしているが、一部互換性のない部分があった。

その後、Classic Mac OSでは、Shift_JISをベースとした文字コードが用いられるようになった。

- **漢字Talk**: 日本語対応の歴史
- **MacJapanese**: 漢字Talkで使われていた文字コード
- **MacJapaneseの問題点とShift_JISへの移行**

macOS (Mac OS X以降)

現在のmacOS（Mac OS X以降）では、内部的にはUnicode（UTF-8）が用いられている。

- **内部エンコーディングとしてのUTF-8**
- **各種エンコーディングへの対応**

macOSにおける日本語入力と文字コード

macOSでは、日本語入力システム（IM）を用いて日本語を入力する。IMは、ユーザーが入力した仮名文字を漢字に変換し、Unicodeのコードポイントに変換する役割を担う。

iOSの文字コード対応

iOSは、Appleが開発するモバイルOSであり、iPhone、iPad、iPod touchなどのデバイスで使用されている。macOSとの親和性が高く、Appleの厳格な品質管理のもとで開発されているため、Unicode対応も非常に高い水準で実現されている。システム全体でUnicodeをサポートしており、さまざまな言語の文字を問題なく表示・入力できる。**内部文字コードとしては、主に**

UTF-16が使用されている。これは、Objective-CやSwiftで主に使用されるNSStringがUTF-16ベースであるためである。 iOSアプリケーションの開発には、Unicodeを前提としたAPIが提供されている。

日本語フォントとしては、ヒラギノ角ゴシック、ヒラギノ明朝、ヒラギノ丸ゴシックなどの高品質なフォントがプリインストールされている。日本語入力システムとしては、フリック入力や音声入力など、使いやすい入力方法が提供されている。

iOSは、Unicodeに準拠しているため、通常の使用で文字化けなどの問題が発生することはほとんどない。ただし、非常に古いシステムや、特定のエンコーディングに依存したシステムとのデータ交換を行う場合は、問題が発生する可能性がある。アプリケーション開発者は、内部的にはUTF-16で文字列を扱うことが多いため、外部システムとの連携ではUTF-8への変換が必要になる場合があることに留意すべきである。

Androidの文字コード対応

Androidは、Googleが中心となって開発するモバイルOSであり、多様なデバイスで使用されている。オープンソースであり、カスタマイズ性が高いことが特徴である。システム全体でUnicodeをサポートしているが、デバイスやアプリケーションによって、文字コード対応のレベルに差がある場合がある。Java言語で開発されたアプリケーションが多く、**Javaは内部的に文字列をUTF-16で扱うため、Androidの内部文字コードもUTF-16が基本となっている。ただし、OSの低レベルAPIや、NDK（Native Development Kit）を使ってC/C++で開発されたアプリケーションでは、UTF-8が使われることも多い。**

日本語フォントは、デバイスメーカーによって異なるが、Noto Sans CJK JP、Roboto、モトヤLシーダ3等、Unicodeに対応したフォントが広く使用されている。日本語入力システムは、Gboard (Google 日本語入力)、Simeji など、さまざまな入力システムが利用できる。

Androidは、デバイスやアプリケーションによって、文字コード対応のレベルに差があるため、一部の環境で文字化けなどの問題が発生する可能性がある。特に、古いデバイスや、日本語対応が不十分なアプリケーションを使用する場合は、注意が必要である。アプリケーション開発者は、文字エンコーディングをUTF-8に統一し、さまざまな環境でのテストを行うことが重要である。また、Javaの文字列が内部的にUTF-16であるため、外部システムとの連携ではエンコーディングの変換が必要になる場合があることに留意すべきである。

ChromeOSの文字コード対応

ChromeOSは、Googleが開発するLinuxベースのOSであり、Webブラウザである Chromeを中心に設計されている。Web標準で広く使用されているUTF-8を基本としており、システム全体でUnicodeをサポートしているため、さまざまな言語の文字を問題なく表示・入力できる。**内部文字コードとしては、Linuxカーネルや多くのライブラリでUTF-8が使用されている**。Chromeブラウザのエンジンは、HTMLやCSSで指定された文字エンコーディングを正しく解釈する。また、Androidアプリケーションの実行環境としても機能するため、Androidの文字コード対応も基本的に継承している。日本語フォントとしては、Noto Sans CJK JP、M+ FONTSなど、Unicodeに対応したフォントがプリインストールされている。日本語入力システムとしては、Google 日本語入力が利用できる。

ChromeOSは、基本的にエンコーディングの自動検出に依存しているため、ユーザーが文字コードを意識する場面は少ない。ただし、古いシステムや特殊な環境とのデータ交換を行う場合は、文字コードの違いによる問題が発生する可能性はゼロではない。開発者としては、Webページやアプリケーションを作成する際に、文字エンコーディングをUTF-8に統一し、HTMLやCSSで正しく指定することが重要である。

外字の実装

標準の文字コード規格に含まれない文字を、**外字**と呼ぶ。外字は、人名や地名などの固有名詞を表記する際や、特殊な記号を扱う際に必要となる。ここでは、外字を実装するための、いくつかの方法について説明する。

外字とは：外字の必要性と問題点

外字とは、標準の文字コード規格に含まれない文字のことである。外字が必要となる主な理由として、以下のようなものが挙げられる。

- 人名や地名などの固有名詞に、標準の文字コード規格に含まれない漢字が使用されている場合
- 特殊な記号やマークを表記する必要がある場合
- 学術研究などで、特殊な文字を扱う必要がある場合

外字を用いることの問題点は、異なる環境間での互換性が確保できないことである。ある環境で作成した外字は、他の環境では正しく表示されない可能性がある。また、外字を含む文書を別のシステムに移行する際に、文字化けが発生するリスクがある。

汎用機における外字の実装

汎用機時代、各ベンダーはそれぞれのメインフレームで用いるために独自の外字を実装していた。これらは、標準の文字コード規格には含まれない、各社のシステム固有の文字であった。

- **IBM**: **IBM漢字コード**に**IBM漢字拡張文字**を追加。
- **富士通**: **JEFコード**に**JEF拡張外字**を追加。
- **日立**: **KEISコード**に**KEIS外字**を追加。
- **NEC**: **JIPSコード**に**JIPS拡張外字**を追加。
- **日本電信電話公社**: **DIPS漢字コード**に独自の外字を追加。

これらの外字は、各社のシステム内では問題なく使用できたが、異なるシステム間でのデータ交換においては、文字化けなどの原因となった。このことが、後の統一的な文字コード規格の必要性へとつながっていく。

Windowsにおける外字の実装

Windowsでは、主に以下の方法で外字を実装・利用することができる。

外字エディター (EUDC)

Windowsに標準搭載されている外字エディター(EUDC)を用いることで、ユーザーが自由に外字を作成・登録し、システムで使用することができる。

- **仕組み**: 外字エディターで作成した外字は、特定のフォント（外字ファイル）と関連付けられ、Unicodeの私用領域 (U+E000 から U+F8FF) のコードポイントに割り当てられる。

- **フォントリンク**: 作成した外字を、特定のフォントや全てのフォントに関連付ける（リンクさせる）ことができる。これにより、関連付けられたフォントを使用するアプリケーションで外字を表示できるようになる。フォントリンクの設定は、レジストリを直接編集するか、外字エディターの「フォントのリンク」機能から行える。

Windowsの標準の文字セットを利用する

Windowsでは、標準でいくつかの文字セットをサポートしており、これらの文字セットには、日本語環境でよく使われる外字が含まれている場合がある。

- **機種依存文字**: シフトJISベースの文字セット(CP932)に含まれる、NEC特殊文字、NEC選定IBM拡張文字、IBM拡張文字など。これらはUnicodeにも一部が収録されているが、互換性維持のために残されているコードポイントであり、使用は推奨されない。

- **Unicodeのサロゲートペア**: Unicodeの基本多言語面 (BMP) に収まらない文字を、サロゲートペアと呼ばれる2つのコードポイントの組み合わせで表現する方法。例えば、人名用漢字の一部は、サロゲートペアで表現される。サロゲートペアは外字ではないが、その技術が前提とする追加面（Supplementary Planes）の利用が一般化したことにより、これまで「外字」として扱わざるを得なかった多くの文字が、標準のUnicode符号位置で表現可能となった。これにより、システムごとに異なる外字を管理・交換する必要性が低減し、データの互換性や流通性を高める上で大きな前進となった。

サードパーティ製の外字フォントを利用する

市販の外字フォント製品を導入することで、多数の外字を利用することができる。これらの製品には、人名や地名、記号など、特定の用途に特化した外字が収録されていることが多い。

- **利点**: 高品質な外字が多数利用できる。
- **欠点**: 有償であることが多い。また、フォントがインストールされていない環境では、外字が表示できない。

Windows NT(R) 漢字処理技術協議会 (XKP)

1996年、Windows NT 4.0のリリースに合わせ、**Windows NT(R) 漢字処理技術協議会 (eXtended Kanji Processing、略称：XKP)** が発足した。XKPは、日本アイ・ビー・エム、富士通、日立製作所、日本電気、三菱電機、東芝、沖電気工業、およびマイクロソフトの8社で構成され、Windows NT環境における漢字処理の問題解決と標準化を目的として活動した。

XKPの主な取り組みとして、以下が挙げられる。

- **標準化された外字環境の提供**: XKPは、外字の標準的な取り扱い方法を策定し、Windows NT 4.0に**標準外字セット（XKP標準外字）**として実装した。これにより、異なるベンダー間での外字の互換性問題の軽減を図った。XKP標準外字は、外字エディターで作成する外字とは異なり、特定のベンダに依存しない外字セットとして提供された。

- **13社の合意による「統一外字」の制定**: XKP参加8社にキヤノン、シャープ、リコー、松下電器産業、カシオ計算機を加えた13社で「統一外字」の制定を進め、約3,000文字の統一外字を定義した。

- **Unicode 対応の推進**: XKP は、Unicode の普及を推進し、Windows NT 環境における Unicode 対応の基盤を整備した。

XKP の活動は、Windows における日本語環境の改善に大きく貢献したが、外字問題の根本的な解決には至らなかった。特に、XKP 標準外字と外字エディターで作成する外字との間の互換性問題や、異なるバージョンの Windows 間での外字の非互換性の問題は残された。

Windowsにおける外字の課題

Windows で外字を利用する際には、以下の点に注意する必要がある。

- **互換性**: 外字エディターで作成した外字や、サードパーティ製の外字フォント、XKP 標準外字は、異なる環境間での互換性が保証されない。特に、Web アプリケーションなど、不特定多数のユーザーがアクセスする環境では、外字の使用は避けるべきである。
- **文字化け**: 機種依存文字は、異なる OS やアプリケーション間でのデータ交換において、文字化けの原因となる可能性が高い。
- **フォントリンクの管理**: 外字エディターで作成した外字を複数のフォントに関連付ける場合、フォントリンクの設定が煩雑になることがある。

Unicodeの私用領域を利用した外字

Unicode では、ユーザーが自由に使える**私用領域**（Private Use Area）が用意されている。私用領域は、U+E000 から U+F8FF までの範囲（基本多言語面）、および U+F0000 から U+FFFFD までと U+100000 から U+10FFFD までの範囲（追加私用面）に割り当てられている。

私用領域を用いることで、標準の文字コード規格に含まれない文字を表現することができる。ただし、私用領域の使用方法は、ユーザーやシステムに委ねられているため、異なる環境間での互換性は保証されない。

IVS/IVDを利用した外字

IVS（Ideographic Variation Sequence）は、漢字の異体字を表現するための仕組みである。IVS では、基底文字と異体字セレクタ（Variation Selector）を組み合わせて、異体字を表現する。異体字セレクタは、**IVD**（Ideographic Variation Database）というデータベースで管理されている。

IVS を用いることで、標準の文字コード規格に含まれない異体字を、互換性のある方法で表現することができる。ただし、IVS に対応したフォントや入力システムが必要となる。

Webフォントを利用した外字

近年、Webアプリケーションの普及に伴い、**Webフォント**を利用して外字を表示するケースが増えている。Webフォントとは、Webサーバー上に配置されたフォントデータを、Webブラウザがダウンロードして表示する仕組みである。

Webフォントの仕組みと特徴

- **仕組み**: Webフォントは、CSSの`@font-face`ルールを用いて定義する。`@font-face`ルールでは、フォントの名前、フォントファイルのURL、フォントのスタイルなどを指定する。Webブラウザは、`@font-face`ルールで指定されたフォントファイルをダウンロードし、そのフォントを用いてテキストを表示する。外字を含むWebフォントを利用することで、ユーザーの環境に特定のフォントがインストールされていなくても、外字を表示することができる。

Webフォントの利点

- **環境非依存**: ユーザーの環境に依存せずに、外字を表示することができる。これにより、異なるOSやブラウザ間での表示の差異を最小限に抑えることができる。
- **容易な管理**: 外字を含むフォントデータをWebサーバー上で一元管理することができる。
- **デザインの統一**: Webサイトのデザインに合わせて、外字を含むフォントを自由に選択することができる。

Webフォントの欠点

- **パフォーマンス**: フォントファイルのダウンロードに時間がかかる場合、テキストの表示が遅れる可能性がある。特に、多数の外字を含む大きなフォントファイルを使用する場合は、注意が必要である。
- **ライセンス**: 商用フォントを利用する場合、Webフォントとしての利用がライセンスで許可されていることを確認する必要がある。
- **セキュリティ**: 信頼できないソースから提供されたWebフォントを利用すると、セキュリティ上のリスクが生じる可能性がある。
- **表示の差異**: ブラウザやOSによって、Webフォントのレンダリング結果に若干の差異が生じる可能性がある。

Webフォント利用上の注意点

- **サブセット化**: 外字を含むフォントファイルは、サイズが大きくなる傾向がある。パフォーマンスを向上させるために、必要な文字のみを含むように**サブセット化**することが推奨される。サブセット化とは、フォントファイルから不要なグリフ（字形データ）を除去し、ファイルサイズを削減する技術である。
- **フォールバック**: Webフォントのダウンロードに失敗した場合や、ブラウザがWebフォントに対応していない場合に備えて、**フォールバック用のフォント**を指定することが重要である。フォールバック用のフォントは、`font-family`プロパティで、Webフォントの後に指定する。
- **適切なフォーマットの選択**: Webフォントには、WOFF、WOFF2、TrueTypeなど、いくつかのフォーマットがある。ブラウザの対応状況やファイルサイズなどを考慮して、適切なフォーマットを選択する必要がある。一般的には、WOFF2が圧縮率が高く、多くのモダンブラウザでサポートされているため、推奨される。

ユニコードでの開発

プログラミング言語の文字コード対応

プログラミング言語における文字コードの扱いは、その言語処理系が提供する機能、実行環境のOSのロケール、利用可能なライブラリ、そして開発者が手動で実装する範囲によって大きく異なり、多岐にわたる。

言語処理系そのものに文字コード対応が組み込まれている場合、開発者は比較的容易に文字コードを扱うことができる。一方、OSのロケールに依存する場合は、異なる環境での動作に注意が必要となる。また、ライブラリを利用する場合は、そのライブラリの機能と制限を理解する必要がある。

さらに、アプリケーション内で手組みで実装する場合は、文字コードに関する深い知識に基づいて、複雑な処理を自ら記述する必要がある。このように、文字コードへの対応は様々なレベルで存在し、それぞれのレベルで異なる課題が存在する。

本稿では、主要なプログラミング言語における文字コード対応の状況を、サロゲートペアやIVS/IVDの情報も含めて説明する。

C/C++

CおよびC++は、文字コードに対する直接的な組み込みサポートは比較的低レベルである。文字コードの扱いは、使用する文字型（`char`, `wchar_t`, `char16_t`, `char32_t`）や、利用するライブラリに大きく依存する。

対応文字コード

- `char`: 通常は1バイトで、ASCIIやISO-8859-1などのシングルバイトエンコーディング、またはUTF-8のコードユニットを格納するために使用される。
- `wchar_t`: ワイド文字型で、処理系によってサイズが異なる（Windowsでは通常2バイト、Unix系では4バイト）。WindowsではUTF-16、Unix系ではUTF-32を格納することが一般的である。
- `char16_t`: C++11で導入された16ビット文字型で、UTF-16のコードユニットを格納するために使用される。

- `char32_t`: C++11で導入された32ビット文字型で、UTF-32のコードポイントを格納するために使用される。
- **ライブラリ:** 文字コード変換には、標準ライブラリの`<locale>`ヘッダや、iconvライブラリ、ICU (International Components for Unicode) ライブラリなどがよく用いられる。

サロゲートペア

- `wchar_t` **(Windows):** UTF-16を使用するため、サロゲートペアを扱うことが可能である。
- `char16_t`: UTF-16のコードユニットを格納するため、サロゲートペアを扱うことが可能である。
- `char32_t`: UTF-32でコードポイントを直接表現するため、サロゲートペアの概念は意識する必要がない。
- `char` **(UTF-8):** UTF-8エンコーディングのルールに従って、複数の`char`でサロゲートペアで表現されるコードポイントを表現できるが、直接的なサポートはライブラリに依存する。

IVS/IVD対応

- C/C++自体にはIVSを直接扱う機能はない。
- **ライブラリ:** ICUライブラリを利用することで、IVSを扱うことが可能である。ICUは、Unicode標準に準拠した高度なテキスト処理機能を提供する。

Java

Javaでは、内部的にはUnicode（UTF-16）が用いられている。Javaの文字列（`String`クラス）は、UTF-16でエンコードされた文字列を保持する。

対応文字コード

- **基本:** UTF-16 (内部表現)である。
- **入出力:** `InputStreamReader`や`OutputStreamWriter`クラスを利用することで、Shift_JIS、EUC-JP、UTF-8など、様々な文字コードとの間で変換が可能である。`Charset`クラスでサポートされているエンコーディングを確認できる。

サロゲートペア

- Javaの`String`クラスは、UTF-16のサロゲートペアを正しく扱える。

- `Character`クラスの`isHighSurrogate()`、`isLowSurrogate()`、`toCodePoint()`などのメソッドを使って、サロゲートペアを構成するコードユニットや、サロゲートペアで表現されるコードポイントを操作することが可能である。
- **バージョン:** Java 5以降で完全にサポートされている。

IVS/IVD対応

- Java 9以降、IVSに対応するためのAPIが導入された。
- `Character`クラスの`UnicodeBlock.IDEOGRAPHIC_VARIATION_SEQUENCES`定数でIVS関連のブロックが定義されている。
- `String`クラスや`Character`クラスのメソッドを使ってIVSを含む文字を扱うことが可能である。
- **ライブラリ:** ICU4Jなどのライブラリを利用することで、より高度なIVSの処理が可能である。

C# (.NET Framework / .NET)

C#は、.NET Frameworkまたは.NET上で動作するプログラミング言語であり、内部的にはUnicode（UTF-16）が用いられている。C#の文字列（`string`型）は、UTF-16でエンコードされた文字列を保持する。

対応文字コード

- **基本:** UTF-16 (内部表現)である。
- **変換:** `System.Text.Encoding`クラスを用いて、UTF-8、Shift_JIS、EUC-JPなど、様々な文字コードとの間で変換を行うことが可能である。`Encoding.UTF8`や`Encoding.GetEncoding("shift_jis")`などでエンコーディングオブジェクトを取得する。

サロゲートペア

- C#の`string`型は、UTF-16のサロゲートペアを正しく扱える。
- `char.IsHighSurrogate()`、`char.IsLowSurrogate()`、`char.ConvertToUtf32()`などのメソッドを使って、サロゲートペアを操作することが可能である。
- **バージョン:** .NET Framework 2.0以降（.NET Core/5以降を含む）で完全にサポートされている。

IVS/IVD対応

- .NET Framework 4.6以降、IVSに対応するための機能が強化された。
- `System.Globalization.StringInfo`クラスを使って、IVSを含む文字のテキスト要素を列挙したり、長さを取得したりすることが可能である。
- **ライブラリ:** ICU4Nなどのライブラリを利用することで、より高度なIVSの処理が可能である。

Python

Python 3では、内部的にはUnicodeが用いられている。具体的なエンコーディングは実装に依存するが、一般的にはUTF-8が用いられる。Python 3の文字列（`str`型）は、Unicodeコードポイントのシーケンスとして扱われる。

対応文字コード

- **基本:** Unicode (内部表現、多くの場合UTF-8)である。
- **入出力:** `open`関数で`encoding`引数を指定することで、Shift_JIS、EUC-JP、UTF-8など、様々な文字コードでエンコードされたファイルを読み書きすることが可能である。
- **標準ライブラリ:** `codecs`モジュールを利用することで、より細かいエンコーディング/デコーディングの制御が可能である。

サロゲートペア

- Python 3の`str`型は、Unicodeコードポイントを直接扱うため、サロゲートペアを意識する必要はほとんどない。内部的に適切に処理される。
- `ord()`関数で文字のコードポイントを取得したり、`chr()`関数でコードポイントから文字を作成したりする際に、サロゲートペアで表現されるコードポイントも扱える。
- **バージョン:** Python 3以降で完全にサポートされている。

IVS/IVD対応

- Python 3では、IVSも他のUnicode文字と同様に扱うことが可能である。
- **ライブラリ:** `unicodedata`モジュールを使って、文字のプロパティ（名前、カテゴリなど）を取得できるが、IVSを直接操作する機能は限定的である。より高度な処理には、`regex`ライブラリ（`\X`を使用して結合文字シーケンスを扱う）や、より専門的なテキスト処理ライブラリが必要になる場合がある。

Ruby

Rubyにおける文字列は、CSI（Code Set Independent）モデルを採用しており、各文字列オブジェクトが独自のエンコーディング情報を保持している。これはPythonなどの「単一エンコーディング」方式とは異なるアプローチである。

対応文字コード

基本:

- デフォルトエンコーディングはUTF-8である
- 各文字列オブジェクトが独自のエンコーディング情報を保持する
- 95種類以上の文字エンコーディングをサポートしている

入出力:

- `File.open`メソッドでエンコーディングを指定可能である
- 外部エンコーディングと内部エンコーディングを個別に指定できる
- 標準入出力のエンコーディングは環境変数で制御可能である

標準ライブラリ:

- `String`クラスに豊富なエンコーディング変換メソッドが実装されている
- `Encoding`クラスでエンコーディングの制御が可能である
- `StringIO`クラスで文字列の入出力をストリームとして扱える

サロゲートペア

- Rubyの文字列はUTF-8として内部的に保持されるため、サロゲートペアを意識せずに扱える
- `String#length`メソッドは文字数を、`String#bytesize`メソッドはバイト数を返す
- 文字列の分割や結合時には自動的にサロゲートペアが考慮される

IVS/IVD対応

- IVSを含む文字列を直接扱うことが可能である
- 文字列の正規化には`unicode_normalize`メソッドを使用する
- 正規表現でUnicodeプロパティを使用して文字クラスを指定できる
- IVSを含む文字列の比較や検索には適切な正規化処理が必要である

このように、Rubyの Unicode 対応は広範かつ柔軟であるが、特定のケースでは適切な処理方法の選択が必要となる。

JavaScript / TypeScript

JavaScript は現代のソフトウェア開発において不可欠な言語であり、ブラウザだけでなく、Node.js、Deno、Electron、Tauri など様々な実行環境で広く採用されている。TypeScript は JavaScript に静的型付けを追加した上位互換言語であり、大規模アプリケーション開発での採用が進んでいる。

実行環境と文字コード処理

内部的にはすべての環境で UTF-16 が採用されており、文字列処理の一貫性が保たれている。外部とのインターフェースでは UTF-8 が標準的に使用される。Unicode 対応が言語仕様レベルで実装されているため、国際化対応アプリケーションの開発において有力な選択肢となっている。

環境別の特徴

Node.js では `fs` モジュールを通じて様々な文字エンコーディングをサポートし、Deno では `TextEncoder` / `TextDecoder` API を活用した柔軟な文字コード変換が可能である。デスクトップアプリケーション開発においては、Electron と Tauri という 2 つの選択肢がある。Electron は純 JavaScript 環境を提供し、Tauri は Rust バックエンドと組み合わせることで高効率な実行環境を実現している。

Unicode 対応の最新状況

サロゲートペアの処理
最新の JavaScript 実装では、文字列操作メソッドの多くがサロゲートペアを適切に処理できるように改善されている。

```
const emoji = "🌍";
console.log(emoji.length); // 2（サロゲートペア）
console.log([...emoji].length); // 1（文字単位）
```

IVS（異体字セレクタ）対応
Unicode 15.1 までの異体字セレクタに対応し、以下の機能が利用可能である：

- `Intl.Segmenter`による文字単位の分割
- `String.prototype.normalize()`によるIVS正規化
- カスタムフォントによる異体字表示

2024年10月にリリースされたDeno 2.0では、Node.jsとの完全な後方互換性を実現し、npmパッケージの利用が可能となった。TypeScript 5.0以降では、文字列操作に関する型定義が改善され、サロゲートペアやIVSを考慮したより安全なコード記述が可能になっている。

```
function processEmoji(text: string): string {
  return Array.from(text).reverse().join('');
}
```

また、各実行環境では、ICU（International Components for Unicode）ライブラリの最新版が組み込まれ、より豊富なUnicode機能が利用可能になっている。

Rust

Rustは、文字列をUTF-8でエンコードすることを基本としている。`String`型と`&str`型は、有効なUTF-8シーケンスを保持することが保証されている。

対応文字コード:

- **基本:** UTF-8 (内部表現)である。
- **変換:** `std::str::from_utf8`などでUTF-8バイト列から`String`への変換が可能である。他のエンコーディングとの変換には、`encoding_rs`クレートなどの外部ライブラリを利用する。

サロゲートペア:

- Rustの`String`と`&str`はUTF-8でエンコードされているため、サロゲートペアを直接扱う必要はない。UTF-8として正しくエンコード・デコードされる。
- `char`型はUnicodeスカラー値を表し、サロゲートコードポイントは`char`型としては表現できない。
- 文字列のイテレーションはコードポイントごとに行われるため、サロゲートペアも正しく処理される。

IVS/IVD:

- Rustの標準ライブラリでは、IVSを直接的に操作する機能は限定的である。
- **ライブラリ:** より高度なIVS処理には、`unicode-segmentation`クレートや、より専門的なテキスト処理ライブラリが必要になる場合がある。

COBOL

COBOLは、事務処理用に開発されたプログラミング言語であり、レガシーシステムで広く用いられてきた。COBOLでは、言語仕様としては特定の文字コードを規定していないが、実際には、各ベンダーが提供するコンパイラによって、扱い可能な文字コードが異なる。さらに、既存のコードベースや技術者のスキルを考慮すると、文字コード対応には特有の課題が存在する。

対応文字コード:

- **従来:** EBCDIC (メインフレーム)、ASCII (UNIX系など) など、ベンダーや環境に依存する。日本語環境ではShift_JISやEUC-JPも一般的であった。これらの環境では、文字コードはコンパイラや実行環境の設定に強く依存し、プログラム自体で明示的に文字コードを扱うことは少なかった。
- **近年:** 近年のCOBOL処理系では、Unicode（UTF-8、UTF-16）をサポートするものも登場している。しかし、既存のシステムとの互換性を保つため、従来の文字コードも引き続きサポートされていることが多い。

サロゲートペア:

- **古い処理系:** サロゲートペアの概念がないか、正しく扱えない可能性がある。バイト単位での処理が主流であり、マルチバイト文字も特定の規則に従って扱われていた。サロゲートペアのような高度な概念は考慮されていないことが多い。
- **新しい処理系:** Unicode対応の処理系であれば、サロゲートペアを扱うことが可能である。しかし、既存のコードがサロゲートペアを考慮していない場合、予期せぬ問題が発生する可能性がある。具体的なサポート状況はコンパイラのマニュアルを参照する必要がある。

IVS/IVD:

- **古い処理系:** IVSの概念がないため、正しく扱うことはできない。文字は特定のコードポイントとして固定的に扱われ、異体字セレクタの概念は存在しない。

- **新しい処理系:** Unicode対応の新しい処理系では、IVSを扱うことができる可能性があるが、サポートは限定的かもしれない。ベンダー提供のドキュメントを確認する必要がある。既存のコードベースでは、IVSを区別する処理が実装されていることは稀であり、対応には大幅な改修が必要となる可能性がある。

既存のコードベースと技術者のスキルを活かそうとした場合の課題

COBOLの文字コード対応においては、単に処理系の機能だけでなく、長年運用されてきた既存のコードベースと、それを支えてきた技術者のスキルという側面も重要となる。

- **文字コードの暗黙的な前提:** 既存のCOBOLアプリケーションは、特定の文字コード（例えばShift_JIS）でデータが格納されていることを前提に開発されている場合が多い。プログラム中で文字コードが明示的に扱われていないことも多く、文字コードの変更は広範囲に影響を及ぼす可能性がある。
- **バイト単位の処理:** 古いCOBOLアプリケーションでは、文字列をバイト列として扱い、特定のバイト数でフィールドの長さを定義していることがある。Unicodeのような可変長エンコーディングへの移行は、既存のデータ構造や処理ロジックに大きな変更を強いる可能性がある。
- **技術者のスキルギャップ:** 長年COBOL開発に携わってきた技術者は、Unicodeやサロゲートペア、IVSといった比較的新しい概念に精通していない場合がある。新しい技術を取り入れるためには、技術者の再教育やトレーニングが必要となる。
- **移行の複雑さ:** 既存のCOBOLシステムをUnicodeに対応させるには、データ変換、プログラム改修、テストなど、多岐にわたる作業が発生する。これらの作業は、システムの規模によっては非常に大規模かつ複雑なプロジェクトとなる可能性がある。
- **ベンダー依存:** COBOL処理系の文字コード対応状況はベンダーに依存するため、特定のベンダーの製品に特化した知識や対応が必要となる場合がある。

既存のCOBOL資産を活かしつつ、文字コード対応を進めるためには、既存システムの詳細な分析、段階的な移行戦略、そして技術者のスキル向上への投資が不可欠である。安易な移行は、システム全体の安定性を損なうリスクを伴うため、慎重な検討が求められる。

バッチファイル（Windows）

バッチファイルは、Windows環境で利用されるスクリプト言語であり、コマンドプロンプト上で実行される。文字コードの扱いは、実行環境のコードページ設定に依存する。

対応文字コード:

- **基本:** 実行環境のコードページ設定に依存する。日本語環境のEUC環境においては、一般的にShift_JIS (CP932) が用いられることが多い。ANSIコードページとも呼ばれる。

サロゲートペア:

- **非対応:** 標準のバッチファイル処理では、サロゲートペアを正しく扱うことはできない。バッチファイルは通常、シングルバイトまたはダブルバイト文字コードを前提としており、サロゲートペアのような4バイトで表現される文字は文字化けするか、正しく認識されない。

IVS/IVD:

- **非対応:** IVS/IVDにも対応していない。バッチファイルは文字を個々のコードポイントとして認識する能力がなく、異体字セレクタを伴う文字を区別することはできない。

EUC環境における対応:

EUC環境でバッチファイルを用いてサロゲートペアやIVS/IVDを扱うことは、標準機能では非常に困難である。これらの文字を扱う必要がある場合、PowerShellなどのより高機能なスクリプト環境への移行を検討する必要がある。PowerShellはUnicode（UTF-8）をネイティブにサポートしており、サロゲートペアやIVS/IVDの処理が可能である。ただし、PowerShellの利用がEUC環境で許可されているか、既存のシステムとの連携が可能かといった点は別途検討が必要となる。

バッチファイルは、ファイルの中身を単なるバイト列として扱う分には、IVSが含まれていても特に問題は発生しないと考えられる。例えば、`type`コマンドでIVSを含むテキストファイルの内容を表示したり、`copy`コマンドでファイルを複製したりするだけであれば、IVSが何であるかを意識せずに処理できる。

ただし、以下の点に注意すれば、限定的に対応できる場合もある。

- **ファイル名:** IVSを含むファイル名を直接バッチファイルで扱う場合、実行環境のコードページ（EUC環境であればShift_JIS）によっては、ファイル名を正しく認識できない可能性がある。ファイル名を直接指定するのではなく、ワイルドカード (`*`, `?`) を利用したり、別の方法でファイルを特定したりする必要があるかもしれない。例えば、`for`ループでファイルを列挙し、ファイル名ではなく処理順序で操作するなどの工夫が考えられる。

- **ファイル内容の解釈:** バッチファイル内でファイルの内容を文字列として解釈しようとすると、コードページの影響を受けるため、IVSは正しく表示・処理されない。IVSを含むファイルの内容をバッチファイル内で解析したり、文字列操作を行ったりすることは避けるべきである。あくまで、ファイルの内容を「透過的に」扱う場合に限られる。
- **他のツールとの連携:** バッチファイルから外部のツールを呼び出してIVSを含むファイルを処理する場合、そのツールがIVSに対応している必要がある。対応していないツールを呼び出すと、文字化けなどの問題が発生する可能性がある。

重要な注意点:

- 上記はあくまで、バッチファイルがIVSを「意識せず」に、バイト列としてファイルを扱う場合に限られる。バッチファイル自体がIVSを理解して処理できるわけではない。
- ファイル名にIVSが含まれる場合、ファイルシステムの操作（名前変更、削除など）が予期せぬ結果になる可能性がある。
- EUC環境では、基本的にShift_JISでファイル名が扱われるため、IVSを含むファイル名を扱う際には、文字コードに関する深い理解と注意が必要となる。

PowerShell

PowerShellは、Windows環境で利用される、より高機能なスクリプト言語およびコマンドラインシェルである。バッチファイルと比較して、文字コードの扱いにおいてより柔軟性と強力な機能を提供する。

対応文字コード:

- **基本:** PowerShellはUnicodeをネイティブにサポートしており、デフォルトではUTF-8が用いられる。パイプラインを流れるオブジェクトの文字列プロパティなども、UTF-8でエンコードされる。
- **入出力:** `Get-Content`や`Set-Content`などのコマンドレットを使用する際、`-Encoding`パラメータで様々な文字コードを指定してファイルの読み書きを行うことが可能である。UTF-8、UTF-16 (BigEndian, LittleEndian)、ASCII、Big5、Unicode (システムの既定のANSIコードページ)、OEM (システムの既定のOEMコードページ) などがサポートされている。

サロゲートペア:

- **対応:** PowerShellはUnicodeベースであるため、サロゲートペアを正しく扱うことができる。文字列の長さの取得 (`$string.Length`) や、文字列操作 (`Substring()`, `Replace()`) なども、サロゲートペアを1文字として正しく処理する。

IVS/IVD:

- **対応:** PowerShellはIVS/IVDを含むUnicode文字を区別して扱うことができる。例えば、IVSを含む文字列の長さを正しく取得したり、IVSを保持したまま文字列操作を行うことが可能である。
- **Culture:** `Get-Culture`コマンドレットで現在のカルチャ情報を確認でき、これが文字コードの扱いに影響を与える場合がある。

EUC環境における対応:

EUC環境においてPowerShellを利用する場合、いくつかの考慮事項がある。

- **文字コード変換:** EUC環境で一般的なShift_JISなどの文字コードでエンコードされたファイルを扱う場合、`Get-Content`コマンドレットの`-Encoding`パラメータで適切なエンコーディングを指定する必要がある。同様に、Shift_JISでファイルを保存する場合は、`Set-Content`コマンドレットで`-Encoding`パラメータを指定する。
- **パイプライン処理:** パイプラインで異なる文字コードのデータが流れる場合、意図しない文字化けが発生する可能性がある。必要に応じて、`[System.Text.Encoding]`クラスを利用して明示的な文字コード変換を行う必要がある。
- **外部コマンドとの連携:** PowerShellからバッチファイルや他の外部コマンドを実行する場合、それらのコマンドが期待する文字コードを考慮する必要がある。

バッチファイルとの比較:

PowerShellは、バッチファイルと比較して、文字コードの扱いにおいて大きな利点を持つ。Unicodeをネイティブにサポートしているため、サロゲートペアやIVS/IVDを標準機能で扱うことができる。また、`-Encoding`パラメータによる柔軟な文字コード指定や、`.NET Framework`の強力な文字コード関連クラスを利用できる点も強みである。

結論:

行政現場で広く使われているEUC環境において、バッチファイルは文字コード対応に限界がある一方、PowerShellはUnicodeをベースとしており、サロゲートペアやIVS/IVDを比較的容易に扱うことができる。EUC環境でこれらの高度な文字コードを扱う必要がある場合、PowerShellは有力な選択肢となる。ただし、既存システムとの連携や、EUC環境の制約事項を考慮した上で、適切な文字コード設定や変換処理を行う必要がある。

VBAマクロ (Microsoft Office)

VBA (Visual Basic for Applications) は、Microsoft Office製品に組み込まれたマクロ言語である。文字コードの扱いは、Officeアプリケーションの内部処理に依存する。

対応文字コード:

- **基本:** VBA内部ではUnicode（UTF-16）が用いられている。Officeアプリケーションで扱う文字列は、基本的にUnicodeで処理される。

サロゲートペア:

- **限定的な対応:** VBA自体はUnicodeを扱うことができるため、サロゲートペアを格納することは可能である。しかし、VBAの文字列操作関数の一部は、サロゲートペアを正しく処理できない場合がある。例えば、`Len()`関数は文字数ではなく、UTF-16コードユニット数を返すため、サロゲートペアを2文字としてカウントしてしまう。サロゲートペアを正しく扱うには、`StrConv()`関数などを用いて明示的な変換を行う必要がある場合がある。
- **バージョン:** 比較的新しいバージョンのOffice（Office 2010以降など）では、サロゲートペアの扱いが改善されている傾向にある。

IVS/IVD:

- **非対応:** VBAの標準機能では、IVS/IVDを直接的に扱うことはできない。VBAは文字をUnicodeコードポイントとして認識するが、異体字セレクタの概念を理解しないため、IVSを構成するベース文字と異体字セレクタを別々の文字として扱うか、あるいは区別せずに処理してしまう。

EUC環境における対応:

EUC環境でVBAマクロを用いてサロゲートペアを扱うには、いくつかの注意点が必要となる。

- **Officeのバージョン:** Windows 8以降のOffice 2013では、IMEから直接IVS/IVDの入力が可能となり、アドインも不要となった1。一方、Windows Vista/7環境のOffice 2007/2010では、「Unicode IVS Add-in for Microsoft Office」を導入することで約58,000字の異体字を利用できる。このアドインはWord、Excel、PowerPointで利用可能だが、AccessやOutlookには対応していない。
- **フォント:** サロゲートペアを表示するためには、対応したフォントがインストールされている必要がある。

- **文字列操作:** VBAの標準的な文字列操作関数を利用する際には、サロゲートペアが正しく処理されるか確認が必要である。
- **コピー&ペースト:** 異体字データを他のアプリケーションにコピー&ペーストする際は、IVS情報が失われる可能性があることに注意が必要である。

IVS/IVDについては、VBAの標準機能では対応が難しく、もしIVS/IVDを扱う必要がある場合は、外部のAPIやライブラリ（例えば、C++で作成されたDLLをVBAから呼び出すなど）を利用する必要がある。ただし、EUC環境において外部ライブラリの利用が許可されているか、セキュリティ上の問題はないかといった点は慎重に検討する必要がある。

Microsoft Power Platform

Power Platformは、Microsoft 365に含まれるローコード開発プラットフォームであり、Unicode文字コードを標準的にサポートしている。

基本機能と特徴

- Power Platformは以下の主要コンポーネントで構成される：
- Power Apps：業務アプリケーション開発
- Power Automate：業務プロセス自動化
- Power BI：データ分析・可視化
- Power Virtual Agents：チャットボット開発
- Power Pages：Webサイト構築

文字コード処理

基本仕様:

- 内部的にUnicodeを採用し、UTF-8/UTF-16でのデータ処理が可能
- 式言語（Power Fx）では、Unicode文字のシーケンスを直接扱うことが可能

サロゲートペア対応:

- Left、Mid、Right関数では、サロゲートペアを1文字として正しく処理できない
- 文字列長の計算においても、サロゲートペアは2文字としてカウントされる
- データの永続化時にはサロゲートペアは保持される

IVS/IVD対応:

- IVS/IVDを含む文字列は格納可能だが、文字列操作関数での処理は保証されない
- 表示時のフォントはシステムの設定に依存する
- コピー＆ペースト時にIVS情報が失われる可能性がある[3]

開発時の注意点

文字列処理:

- サロゲートペアを含む文字列を扱う場合は、独自の文字列処理関数の実装が必要
- 文字列長の計算には特別な考慮が必要
- IVS文字を含むデータの永続化時は、異体字情報の保持について確認が必要

表示制御:

- IVS対応フォントが必要
- 異体字の表示は環境依存となる
- 文字列の検索や置換時は、基底文字のみで処理される可能性がある

セキュリティと制御

- Azure Virtual Networkとの連携により、通信経路の制御が可能
- Enterprise Policyによる組織全体での一貫した文字コード処理の実現
- データの永続化時の文字コード変換ルールを明確に定義する必要がある

プログラミング言語選択の指針

文字コードの扱いはプログラミング言語によって異なり、サロゲートペアやIVS/IVDへの対応もバージョンや利用するライブラリに依存する。特にレガシーなシステムや、低レベルな処理を行う場合には、文字コードに関する深い理解が必要となる。Unicode（特にUTF-8）への対応は進んでいるが、古い文字コードとの連携が必要な場面も依然として存在する。それぞれの言語や環境における文字コードの扱い方を正しく理解し、適切な処理を行うことが重要である。

EUC環境でこれらの高度な文字コードを扱うには、スクリプト言語の選択、外部ライブラリの利用、またはアプリケーション側の対応が必要となる。しかし、EUC環境の特性（セキュリティポリシー、互換性、既存システムの制約など）を考慮すると、これらの対応は容易ではない場合が多い。現実的には、EUC環境でサロゲートペアやIVS/IVDを扱う必要が生じた場合、影響範囲の精査や業務プロセスの見直しを含めた検討が必要となる可能性が高い。

ミドルウェア選定の留意点

ミドルウェアは、異なるシステム間、あるいはシステム内部のコンポーネント間でデータや処理を連携させる役割を担う。この際、文字コード対応は、データの文字化けや欠落を防ぎ、システムの安定稼働に不可欠な要素である。特定の製品に依存せず、将来にわたり有効なミドルウェア選定における文字コード対応の調査ポイントについて解説する。

製品単位での文字コード対応の重要性

システム連携において、文字コードの不一致は深刻な問題を引き起こす。異なるシステムが異なる文字コードを使用している場合、そのままデータを連携させると文字が正しく表示されず、最悪の場合、データが破損する。故に、ミドルウェアが様々な文字コードに対応し、必要に応じて文字コード変換を適切に行える能力は、重要な評価ポイントとなる。

製品比較にあたっての留意点

ミドルウェアを選定する際には、以下の点を中心に文字コード対応状況を調査・確認する必要がある。

- **対応文字コードの種類:** UTF-8はもとより、EUC-JP、Shift_JISなど多様な文字コードに対応しているかを確認する。過去のシステムとの連携を考慮すれば、幅広い文字コードへの対応が求められる。
- **Unicodeサポートの詳細:**
  - **サロゲートペア:** BMP (基本多言語面) に収まらない文字（例：一部のCJK統合漢字拡張）を扱うためのサロゲートペアに対応しているか。対応がない場合、これらの文字が欠落したり、文字化けしたりする可能性がある。
  - **IVS/IVD:** 漢字の異体字を区別して扱えるか。人名や地名など、固有名詞を正確に扱う必要がある場合に重要となる。対応状況によっては、異体字が同一の文字として扱われてしまう可能性がある。
- **文字コード変換機能:**
  - **対応可能な変換:** 異なる文字コード間での変換を柔軟に行えるか。変換時に情報が失われる可能性についても考慮が必要である。
  - **変換ロジックのカスタマイズ:** 複雑な文字コード変換ルールに対応するために、変換ロジックをカスタマイズできる柔軟性があるか。
  - **変換処理の性能:** 大量のデータを扱う場合に、文字コード変換処理がボトルネックとならないか。性能要件を満たす処理能力があるかを確認する。

- **メタデータにおける文字コード対応:** データ内容だけでなく、テーブル名、カラム名、ファイル名などのメタデータにおいても、文字コードが正しく扱われるか。データとしては扱えても、メタデータで文字化けが発生するケースも存在する。
- **ソート・検索における文字コード対応:** 文字コードを考慮したソートや検索が可能か。例えば、異なる文字コードで表現された同一の文字が、検索時に正しくヒットするか、期待通りの順序でソートされるかを確認する。
- **文字列処理機能:** 異体字のノーマライゼーション、半角・全角変換、大文字・小文字変換など、文字コードを意識した文字列処理機能を備えているか。これらの機能は、データ品質の維持やデータ分析において重要となる。
- **内部的な文字表現:** ミドルウェア内部で文字データをどのように扱っているか。内部表現がUnicodeベースであれば、様々な文字コードを扱いやすいと考えられる。
- **メタデータにおける文字コード情報:** 連携するデータの文字コード情報を正しく保持し、伝播できるか。これにより、後続の処理で適切な文字コード解釈が可能となる。
- **エラーハンドリング:** 文字コード変換時にエラーが発生した場合、どのように処理されるか。エラーログの出力や、エラー発生時の代替処理など、適切なエラーハンドリング機能が備わっているかを確認する。
- **APIやインターフェース:** ミドルウェアが提供するAPIやインターフェースを通じて文字データを扱う場合、文字コードを明示的に指定できるか、または自動的に判別できるか。
- **ドキュメントとサポート:** 文字コード対応に関する詳細なドキュメントが提供されているか。また、文字コードに関する問い合わせに対応できるサポート体制が整っているか。

注意すべき落とし穴

ミドルウェア選定時には、以下の点に注意が必要である。

- **「Unicode対応」の落とし穴:** 単に「Unicode対応」と謳っていても、サロゲートペアやIVS/IVDへの対応状況は異なる場合がある。詳細な仕様を確認することが肝要である。
- **対応レベルの多様性:** 文字コード対応には様々なレベルが存在する。データの格納はできても、ソートや検索で正しく扱えない、あるいはメタデータでは対応していないなど、部分的な対応に留まる場合がある。
- **デフォルト設定の確認:** デフォルトの文字コード設定が適切であるとは限らない。連携するシステムに合わせて設定を変更する必要がある場合がある。
- **テストの重要性:** 実際に様々な文字コードのデータを用いて連携テストを行い、文字化けやデータ欠落が発生しないことを確認することが不可欠である。特に、サロゲートペアやIVS/IVDを含むデータでのテストは必須である。
- **バージョンによる差異:** 同じミドルウェア製品でも、バージョンによって文字コード対応状況が異なる場合がある。導入予定のバージョンで必要な機能がサポートされているかを確認する必要がある。

ミドルウェア選定の指針

ミドルウェアの選定において、文字コード対応は単なる機能の一つではなく、システム全体の信頼性を左右する重要な要素である。表面的な対応状況だけでなく、サロゲートペアやIVS/IVDといった詳細なUnicode仕様への理解と対応が求められる。製品名に依存した情報に惑わされず、上記のチェックポイントを参考に、将来にわたって安心して利用できるミドルウェアを選定することが肝要である。文字コード対応には様々なレベルが存在することを認識し、自社の要件に合致した対応レベルを持つミドルウェアを選択する必要がある。

全面的なUnicode対応が困難な場合

システムの規模や構造によっては、全ての構成要素をUnicodeに完全対応させるのが困難な場合も存在する。特に、長年運用されてきた基幹系システムなどでは、文字コードの変更が広範囲に影響を及ぼし、多大なコストと時間を要する可能性がある。そのような状況下では、段階的な対応や部分的な対応といった現実的なアプローチを検討する必要がある。

- **入出力におけるUnicode対応:** システム内部の文字コードは維持しつつ、ユーザーインターフェースや外部システムとの連携部分でUnicodeを利用する方法が考えられる。例えば、氏名などの文字情報をUnicode対応のデータベースから取得し、画面表示や帳票出力時にのみUnicodeで扱うといった運用である。この場合、システム内部では別の文字コード（例えばShift_JISやJIS X 0208など）で情報を保持し、入出力時に文字コード変換を行うことになる。ただし、この方式では、**システム内部で扱う文字に制限が生じる**こと、および**文字コード変換処理の負荷や変換エラーのリスク**を考慮する必要がある。

- **内部的な別文字コード利用:** システム内部でUnicode以外の文字コードを利用する場合、**どの文字コードを選定するか**が重要となる。将来的な拡張性や、連携する可能性のあるシステムとの互換性を考慮する必要がある。また、**外字管理**をどのように行うかという課題も残る。UnicodeのPrivate Use Area (PUA) を活用する方法もあるが、相互運用性の問題は依然として存在する。

- **段階的なUnicode化:** システム全体を一度にUnicode化するのではなく、影響範囲の小さい部分から段階的にUnicode対応を進める方法である。例えば、新規開発する機能や、改修頻度の高い機能からUnicodeに対応させる。この場合、**新旧システム間のデータ連携における文字コード変換**が重要な課題となる。

段階的対応における留意点

システム内部の完全なUnicode対応が難しい場合、部分的な対応は現実的な選択肢となりうるが、以下の点に留意する必要がある。

- **データの一貫性維持:** システム全体で異なる文字コードが混在する場合、データの整合性を維持するための厳密な管理が必要となる。
- **文字コード変換処理の複雑化:** 入出力時に文字コード変換が必要となる場合、変換処理の実装やテストが複雑になる。変換ロジックの不備は、文字化けやデータ欠損の原因となる。
- **検索・ソート処理への影響:** 異なる文字コードでデータが格納されている場合、検索やソート処理が複雑になる。文字コードを意識した処理を実装する必要がある。
- **異体字の扱い:** Unicode対応のデータベースから文字情報を引く場合でも、システム内部で異体字をどのように扱うかを検討する必要がある。異体字を区別する必要があるか、代表字に統合しても問題ないかなど、業務要件に基づいた判断が求められる。
- **長期的な保守・運用コスト:** 部分的な対応は、一時的なコストを抑えることができるかもしれないが、長期的に見ると、システムの複雑化を招き、保守・運用コストの増大につながる可能性がある。

行政事務標準文字への対応は、単純なUnicode対応以上に検討すべき事項が多い。複数フォントファイルの管理や、システム全体での整合性維持など、技術的な課題は多岐にわたる。システム内部の完全なUnicode対応が難しい場合には、部分的な対応も視野に入れる必要があるが、その際には上記のような留意点を十分に考慮し、将来的な拡張性や保守性も視野に入れた上で、最適なアプローチを選択することが重要となる。

新規開発・再構築におけるUnicodeへの対応

業務システムの新規構築や再構築は、行政事務標準文字への対応をシステム全体で実現するための重要な機会となりうる。新規開発でのUnicode対応にあたっては、以下の点を考慮して設計・開発を進めることを推奨する。

- **Unicode (IVS/IVD) の積極的な採用:** 再構築においては、可能な限りシステム全体でUnicode (特にIVS/IVD) に対応することを強く推奨する。これにより、将来的な文字コード変換の負担をなくし、異体字の区別も標準的な仕組みで実現可能となる。データベースの文字コード設定、アプリケーションの文字コード処理、ライブラリの選定など、全てのレイヤーでUnicode対応を意識することが望ましい。

- **データモデルにおける異体字の扱い:** 行政事務標準文字には多くの異体字が含まれる。データベース設計段階で、異体字を区別して格納するのか、代表字に統合するのかを明確にし、業務要件に合わせたデータモデルを設計する必要がある。異体字を区別する場合、サロゲートペアやIVSを考慮したデータ型を選定し、検索やソート処理における影響も検証が必要となる。

- **文字コード変換処理からの脱却:** 再構築においては、可能な限り文字コード変換処理を排除する設計を目指すことが望ましい。どうしても既存システムとの連携が必要な場合は、変換処理の実装は最小限に留め、変換ロジックは明確にドキュメント化し、テストを徹底することが重要となる。文字コードの対応表だけでなく、異体字の変換ルールも定義する必要がある。

- **フォントレンダリングの標準化:** IVS/IVD に対応する場合、OSやブラウザの標準的なフォントレンダリング機能を利用することが基本となる。ローカルフォントに依存せず、**Webフォント**を積極的に利用することを推奨する。Web フォントを活用することで、スマートフォンやChrome OS といった、従前であれば外字の利用が困難であった環境においても、行政で必要とされる文字を適切に表示することが可能となる。これにより、特定の端末環境に依存することなく、より多くのユーザーに対して正確な情報伝達を実現できる。特に日本語のような大きなフォントファイルについては、表示に必要な文字種のみを抽出する**サブセット化**や、圧縮率の高い**WOFF2形式**の採用を検討することで、データサイズを大幅に削減し、通信負荷を軽減することができる。CSS の `@font-face` ルールなどを活用し、これらの最適化されたフォントファイルを効率的に配信する仕組みを構築することが考えられる。

- **ユーザーインターフェースにおける入力支援:** Unicode IVS/IVD に対応したとしても、全てのユーザーの入力環境が十分に整っているとは限らない。特に、行政事務標準文字には字形の類似した異体字が多数存在するため、ユーザーが意図した文字を正確に入力するには工夫が必要となる。例えば、**文字パレット**を提供し、異体字を含む文字一覧から選択できるようにする、**部首や画数などによる検索機能**を提供する、入力履歴から候補を表示する、あるいは**IVSセレクタ**のような仕組みを実装するなど、ユーザーが容易かつ正確に目的の文字を入力できるような支援策を検討することが重要となる。

- **テストフェーズにおける網羅的な検証:** 開発したシステムが、様々な環境（OS、ブラウザ、フォント）で行政事務標準文字を正しく表示・入力・処理できることを確認するために、十分なテスト期間を確保することが重要となる。特に異体字の扱いについては、考えられる全てのパターンを網羅したテスト計画を立て、実施することが重要となる。

- **既存システムとの連携における段階的移行:** 段階的な再構築を行う場合、新旧システム間でのデータ連携における文字コード変換は、パフォーマンスやデータ整合性に影響を与える可能性がある。連携方式、データ形式、変換処理の実装などを慎重に検討し、十分なテストを実施することが望ましい。可能であれば、API連携など、文字コードに依存しない連携方式を採用することを検討することが望ましい。

アプリケーションの再構築は、将来を見据えたシステム基盤を構築する上で、重要な機会となりうる。行政事務標準文字への対応は、単なる技術的な課題にとどまらず、業務の正確性、相互運用性、そしてユーザーの利便性に直結する重要な要素である。

新規システムの構築にあたっては、できる限り開発者が文字の複雑さを直接的には意識せずに済み、状態遷移の管理やテストケースを最小限に抑えられるよう、標準的な技術スタックの組み合わせによって複雑な文字要件を満たす設計とすることが望ましい。これは、結果として将来の設計負債の蓄積を抑え、データの信頼性確保に繋がるものと期待される。

文字とアイデンティティ管理

多くの外字が氏名、住所、商号から生まれていることからも分かるように、文字とアイデンティティとは密接に結びついている。文字コードの問題は、単なる技術的な課題に留まらず、個人識別の正確性確保、公平なサービス提供、組織の信頼性維持に関わる重大な課題となる。具体的には、外字や異体字、表記揺れによって、本人確認の精度低下や事務誤り、サービス提供における不公平が生じる可能性がある。さらに、データ紐付けの誤りは、意図しない情報提供や個人情報の誤送信などのセキュリティインシデントにつながる恐れがある。

これらの問題は、個人の権利や財産に直接影響を与えるとともに、行政や企業のデジタルトランスフォーメーションの阻害要因となる。本稿では、アイデンティティ管理における文字コード運用について、外字が発生しがちな住所・氏名・商号の文字同定と管理、それぞれの属性情報における典型的な表記揺らぎ問題、突合・検索時の留意点、システム間連携における相互運用性の確保を含めた具体的な対策を解説する。

外字や表記揺れの主な発生要因

氏名の外字の発生

戸籍法施行規則と氏名の制約

自治体住民システムにおける氏名の外字発生には、戸籍法施行規則の制定が深く関わっている。戸籍法施行規則では、戸籍に記載できる文字が「常用平易な文字」という基準で定められた。この基準の解釈を巡っては裁判も発生するなど、運用面で混乱が見られた。特に、同規則施行以前に生まれた人々の氏名については、原則として出生時の漢字がそのまま使用されたため、規則の定める範囲外の文字、すなわち外字が残存する要因となった。

住民基本台帳法の成立と電算化の萌芽

その後、住民登録法から住民基本台帳法への移行が行われ、住民に関する事務が自治省（現在の総務省）に移管された。この法改正を契機に、住民記録システムの電算化が各自治体で進められることとなる。特に中野区における電算化の開始は、全国的なシステム導入の先駆けとなった。しかし、この電算化は国主導のトップダウンではなく、各自治体の判断に委ねられるボトムアップ方式で進められた点が重要である。

初期の電算化における文字コードの限界

電算化が始まった当時、標準的な文字コード体系であった旧JISコードは、人名に使用される漢字を網羅するには不十分であった。そのため、システムで表現できない文字、すなわち外字の登録が各自治体で個別に行われるようになった。これは、当時の技術的な制約からすれば、避けられない措置であったと言える。

戸籍電算化の動向と外字

住民記録システムの電算化が進む一方で、戸籍においても電算化の動きが本格化した。平成6年の戸籍法改正により、戸籍情報システムの導入が可能となり、順次全国の自治体で導入が進められた。この戸籍電算化においても、氏名に使用される漢字の問題は避けて通れなかった。標準的な文字コードで表現できない漢字は外字として登録されることとなった。ただし、戸籍の電算化においては、全ての戸籍が電子化されたわけではない点に留意する必要がある。

電算化されなかった改製不適合戸籍

特に問題となるのは、電算化の対象とならなかった改製不適合戸籍の存在である。これらの戸籍は、過去の戸籍法に基づいて作成されたものであり、現在の電算システムで扱うことが技術的に困難な場合がある。改製不適合戸籍そのものは電子化されていないが、住民基本台帳法に基づき作成される住民票は電算化されている。この際、電算化されていない改製不適合戸籍に記載された氏名が、電算化された住民票に反映される必要があり、その結果として外字が使用されることとなる。つまり、電算化されていない戸籍の情報が、電算化された住民票システムにおいて外字として顕在化するという構造になっている。この電算化されなかった改製不適合戸籍と、それに連動した住民票における外字の使用が課題となっている。

外字の恒常化とその背景

住民記録システムおよび戸籍システムの電算化が進む中で、JISコードの改定などにより、利用できる文字の範囲は拡大された。しかし、過去に登録された外字と新しい文字コード体系における文字との同定作業は、膨大な手間と時間を要するため、必ずしも十分に進展しなかった。さらに、住民票や各種証明書、郵便物の宛名など、住民にとって氏名は自己を特定する重要な情報であり、本来の漢字表記への強いこだわりが存在した。このため、システム上で代替文字に置き換えることへの抵抗感も根強かった。

地域ごとの事情と外字

地域ごとの事情も外字発生の要因となる。歴史的な地名や家系、地域特有の産業や生業に由来する漢字が用いられる場合がある。また、地域社会の慣習や信仰、本家・分家の区別といった文化的背景も影響する。加えて、方言の発音を漢字で表そうとした言語的な背景や、過去の戸籍作成における誤記といった人為的な要因も、外字発生の背景にある。

これらの多様な地域ごとの事情と、戸籍電算化における電算化されなかった改製不適合戸籍の存在が複合的に絡み合い、自治体住民システムにおける氏名の外字問題は、今日に至るまで解消されていない。

住所の異体字と表記揺れ

住所に外字が混ざる理由は多岐にわたる。まず、氏名と異なり住所には制度的な漢字制限が存在しないため、多種多様な漢字が用いられる。1958年4月自治庁自治局長通知では「地名の書き表し方は、さしつかえのない限り当用漢字字体表を用いる。当用漢字字体表以外の漢字についても、当用漢字表の文字に準じた字体を用いてもよい」としている。

1990年のJIS X 0212、2000年のJIS X 0213の作成にあたっては、『国土行政区画総覧』に掲載されている地名漢字は、原則としてすべて収録することが基本方針とされたが、当時はまだ漢字の包摂規準についての明確な定義はなく、異体字か別の文字かの峻別は難しかった。

1962年に住居表示法が施行された後、多くの地域で町字の地名の見直しが行われた際に、町字の文字が当用漢字へと見直された。例えば官庁街のある霞が関について、地名が「霞が関」、駅名が「霞ケ関」となっている表記揺れは有名だが、1967年に住居表示が実施されるまでは地名も「霞ヶ関」だったところ、「ヶ」が当用漢字に含まれなかったため、全国の地名で省略したり、「が」に置き換えるといったかたちで見直されたことに起因している。この「ヶ」は「箇」の漢字を簡略化して「个」と書き、さらにそれが変化して「ヶ」になったとされるが、一般に記号として扱われるため、当用漢字表には含まれなかった。

自治体名称の異体字と表記揺れ

自治体の中には、名称の字形に強いこだわりを持つ例が見られる。例えば宝塚市は「塚」に点があることに拘っている。常用漢字表では「塚」が印刷標準字体とされていることから、JIS X 0208には「塚」が収録されたが、1990年からJIS X 0212に「塚」も収録された。両方の文字が異なる符号位置を持っているが、よく混同されることから検索・突合にあたっては表記揺れに注意する必要がある。

宮城県塩竈市にも、塩竈と塩釜という2つの表記がみられる。竈は塩を煮詰めるカマドをさし、釜の指し示すカマとは全く異なるもので、塩竈市の正式名称は「鹽竈神社」の社号に因む塩竈市だが、難しい字なので市民に強いることはせず、市としても塩釜の表記を広く認めてきた。さらに竈には竈という非印刷標準字体の異体字もあり、いずれも塩竈の漢字変換で候補として出てくる場合がある。

沖縄県那覇市は、歴史的に那覇と、那の横棒が突き出る異体字を用いていたが、2010年の常用漢字表の改訂で「那」が常用漢字表に含まれるようになったことを受けて、那覇市文書取扱規定を改正し、現在は常用漢字表の印刷標準字体を用いるようになった。しかしながら情報システムによっては那覇の異体字を表現するために使われてきた住基ネット統一文字コードのハングル領域のコードが残存し、文字化けを起こす可能性がある。

「現代の文字コード」の章でも紹介したように、東京都葛飾区の「葛」の字は、JIS90字体が「葛」だった時代から歴史的な表記を尊重していたが、JIS2004から正確に表示されるようになった。葛城市は、合併協議の中でパソコン等での申請の利便性を考慮し、JIS90に合わせてJIS90字体の「葛」を採用した。ところが、後のJIS2004では、その字が逆に異体字として扱われることになった。これは、地名の字形に対するこだわりの背景が、一筋縄ではいかないことを示す例といえるだろう。

町字に残存する外字・異体字

特に住居表示未実施地区、とりわけ非起番の小字では、伝統的な文字が今も使用されている場合がある。住所ベースレジストリの整備では、自治体から町字の名称を収集するにあたって、原則としてJIS X 0213の範囲で記載するように依頼しているが、JIS X 0213に収録されていない異体字を用いている場合には、あわせてその表記も記載するように依頼している。

加えて、不動産登記や法人登記では過去の住所も履歴として残ること、特に不動産登記は第三者対抗要件に過ぎず、更新されない場合もあることから、古い情報に外字が残存する場合がある。さらに方書には氏名や商号といった固有名詞が混入し、これらが外字を含むケースもある。

登記における外字

戸籍は個人の身分関係を公的に証明することを目的とする一方、登記は不動産や会社等の権利関係を明確に記録し、取引の安全性を確保することを目的とする。戸籍の除票は古いものが画像として管理されるが、登記においては全てデジタルで管理する必要がある。過去に登記された情報も、その権利関係を証明する上で重要な意味を持つため、例外なく登録する必要がある。

特に商号は、会社を特定する重要な情報であり、過去に使用された商号や、その商号で使用されていた漢字・変体仮名・記号も正確に記録・管理する必要がある。 結果として、過去の登記で使用された文字、例えば古い地名や商号でのみ使われるような文字も、デジタルデータとして扱う必要が生じる。これが、登記システムに登記固有文字として、戸籍に含まれない文字が数多く存在する理由と考えられる。

外字以外の表記揺らぎ

近年、行政手続きや民間サービスにおける個人情報の利用が拡大している。しかし、日本では個人番号の利用が限定的であるため、氏名、住所、生年月日、性別といった基本情報が、個人の識別情報として広く用いられている。

このような状況において、表記揺らぎや紐付け誤りが頻発しており、様々な問題を引き起こしている。異なるシステム間で個人情報を連携する際、わずかな表記のずれが情報喪失や誤った情報処理を招き、サービス利用や行政手続きに支障をきたすことは言うまでもない。

本稿では、個人情報を構成する要素に着目し、それぞれの要素における表記揺らぎや紐付け誤りの原因について現状を分析し、問題点とその解決策について考察する。

健康保険証記号の機種依存文字

レセプト情報・特定健診等情報データベース(NDB)において、2015年度のレセプトと特定健診データの突合率は改修前で25.7%と低く、システム改修後に87.6%まで改善された。この突合率の低さの主要因の一つとして、健康保険証の記号番号における機種依存文字の問題が挙げられる。

NDBでは患者を特定できない形で匿名化を行い、ハッシュIDを付与してデータを格納している。しかし、保険証記号に含まれる丸付き数字や漢数字などの機種依存文字は、コンピュータシステム間で文字化けや表記揺れを引き起こしやすい。これらの文字を正規化せずにハッシュ化したことで、同一人物のデータであっても異なるハッシュIDが生成され、データの突合が困難となっていた。

この問題に対する技術的な解決策として、システム内部での文字列正規化処理が重要である。具体的には入力された文字列を丸数字や機種依存文字から標準的な表現に変換し、システム内部ではUTF-8で一貫して処理する方法が有効である。2022年6月には健康保険組合における外字対応が終了され、JIS規格内の文字での代替表記に統一されることとなった。これによりデータの突合や検索における正確性が向上し、システム間の相互運用性も確保されている。

氏名のフリガナにおける揺らぎ

氏名のフリガナにおける表記揺れは、システム間の相互運用性や本人確認の正確性に大きな影響を与える重要な課題である。拗音・促音の違いは最も一般的な表記揺れの一つであり、「ジュンジロウ」を「ジユンジロウ」と入力するケースや、「ッ」を大書きにするケースが存在する。また、平仮名・カタカナの混在や、「ズ」と「ヅ」のような同音異表記、長音記号「ー」とハイフン「-」の混同なども発生している。

2023年の改正戸籍法により、戸籍システムに氏名のフリガナを追加するにあたっては拗音・促音を区別できることが必須要件となった。これを受けて、住民基本台帳システムでも同様の対応が進められている。システムにおける対応としては、カタカナでの一元管理を行い、拗音・促音を正しく区別して保持することが基本となる。

ただし、データの保存と検索では異なるアプローチが必要である。保存時には拗音・促音を含めて正確に記録し、検索・突合時には「ジュン」と「ジユン」を同一視するなど、表記揺れを許容する正規化処理を行うべきである。これにより、入力ミスや表記の違いによるデータの取りこぼしを防ぎつつ、正確なデータの保持が可能となる。また、長音記号は全角の「ー」に統一し、文字コードはUTF-8で統一的に管理することで、システム間の相互運用性を確保することができる。

銀行口座名義人フリガナの揺らぎ

銀行システムは、メインフレームを利用する従来型システムとオープン系システムが混在しており、文字コードも EBCDIC と UTF-8/Shift-JIS が併存している状況である。行内の顧客情報（CIF）システムにおいても、拗音・促音を正確に記録できるシステムと、システムの制約により「ｼﾞｭﾝｼﾞﾛｳ」を「ｼﾞﾕﾝｼﾞﾛｳ」のように記録せざるを得ないシステムが存在する。

氏名フリガナは本人確認情報として管理される必要があり、2023年の改正戸籍法により拗音・促音を区別できることが必須要件となった。一方、口座名義人フリガナは振込取引における口座特定のための情報として位置づけられ、全銀システムの制約により、拗音・促音には対応しない半角カタカナでの記録が必要となっている。

振込取引においては、口座名義人フリガナは口座を特定するための重要な要素として機能している。このため、氏名フリガナとは別に、全銀システムの制約に準拠した形式で口座名義人フリガナを管理する必要がある。多くの銀行では、口座確認サービスを提供し、振込先の銀行名・支店名・口座番号から自動的に口座名義人を表示する仕組みを実装している。

このように、氏名フリガナと口座名義人フリガナは、その用途と求められる要件が異なることから、システム上も分離して管理することが望ましい。これにより、本人確認の正確性と振込取引の利便性の両立が可能となる。

外国人氏名のフリガナの揺らぎ

また改正戸籍法の施行後も、外国人については氏名の公証されたアルファベットやフリガナの表記が、システムの桁数から溢れて途中で切れてしまうケースや、フリガナが公証されておらず、アルファベット表記をフリガナに当てはめる一般的なルールがないなどの問題がある。

外国人の中には入国当初はカタカナを理解せず、口座開設時に銀行から薦められたフリガナを登録するが、入国から時が経ってカタカナを理解するようになってから表記を自分好みに変更し、銀行口座の開設時期によって口座名義人フリガナに表記揺れが起こり得ること、通称の利用もあることなどから、様々なケースで表記揺れが発生する。

突合・検索における異体字の正規化

氏名・住所における文字の突合・検索においては、異体字の正規化が重要な課題となっている。デジタル庁が提供する abr-geocoder は、アドレス・ベース・レジストリの住所データを正規化する際に、地名に使用される文字に限定して異体字の正規化マップをコード内に組み込んでいるが、氏名等を含めた包括的な異体字対応にはより広範な取り組みが必要である。

異体字には、IVS（Ideographic Variation Sequence）による異体字表現と、独立したコードポイントを持つ異体字の二種類が存在する。前者は基底文字に異体字セレクタを付加することで表現され、後者は「髙」「﨑」「邉」「吉」などの人名異体字が該当する。Unicode 正規化（NFC/NFD）では、CJK 互換漢字と CJK 統合漢字の一部が同一視されるものの、すべての異体字が正規化されるわけではない。

文字情報基盤整備事業では、6 万の漢字を表現できない環境向けに代替漢字へと変換する縮退マップを提供しているが、これは文字コード範囲を狭めるための変換であり、異体字の同一性判定には適していない。そのため、IVS による異体字と独立コードポイントを持つ異体字の両方を考慮した、包括的な異体字正規化マップの整備が必要となる。また、検索時には異体字を同一視

し、保存時には元の文字を保持する二段階の処理方式を採用することが望ましい。このように、文字コードの縮退とは別に、異体字の対応関係を定義した正規化マップを整備することで、より正確な文字の突合や検索が可能となる。

紐付け誤りを起こさないデータ管理へ向けて

本稿では、個人を特定する情報における表記揺らぎや紐付け誤りという、看過できない問題点について論じた。これらの問題は、単なる表記のずれに留まらず、情報の滅失、誤った情報処理などの深刻なインシデントを引き起こす。

これらの問題を解決するためには、以下のような多角的な対策が必要となる。

- **表記揺らぎを吸収する名寄せ技術の導入**: AIや機械学習を活用した名寄せ技術を導入し、表記揺らぎを吸収することで、データの一貫性を確保する必要がある。特に、フリガナの揺らぎ、アルファベット表記の揺らぎ、機種依存文字などの処理は自動化を検討すべきである。

- **システム間連携における標準化の推進**: 各システムが独自の表記ルールを用いるのではなく、標準化されたデータフォーマットおよびAPIを用いることで、情報連携における誤りを防ぐことができる。この際、データ項目の定義を明確化し、相互運用性を高める必要がある。

- **データ入力時の厳格化とデジタル授受の推進**: データ入力時のバリデーションを強化し、誤った情報の登録を防ぐとともに、可能な限り紙媒体を廃止し、デジタルでのデータ授受を推進することで転記ミスの発生を抑制する。この際、入力フォームの最適化や、データ自動補完機能を活用することが望ましい。

- **業務フローの見直し**: データ入力、照合、登録、更新を含む業務フロー全体を見直し、エラー発生箇所を特定し、改善策を講じる必要がある。特に、複数の担当者が関わるフローにおいては、チェック体制を強化し、責任の所在を明確化することが重要となる。

継続的な監視と改善: 導入した対策が有効であるか、継続的に監視し、必要に応じて改善を行う。定期的なデータ監査や、問題発生時の原因分析を実施し、PDCAサイクルを回す必要がある。

今後、これらの対策を総合的に講じることで、より正確かつ効率的な情報連携が実現し、住民サービスの向上に繋がることを期待する。

行政事務標準文字の包摂基準

民間における外字の包摂と、戸籍や住民記録システムにおける包摂は、必ずしも同じ基準である必要はない。しかしながら氏名等を公証する行政機関において、どういった文字を包摂しているかの基準は、他の情報システムにおいても文字運用の指針となり得ると考えられる。

文字包摂ガイドラインは2024年5月に公表され、実際のグリフを元に包摂する字例、包摂しない字例を解説していることから、実際の作業にあたっては直接ガイドライン本体を参照することを推奨する。

文字包摂ガイドラインにおける用語

- **字体** 文字の骨組みであり、個別の文字の形状それぞれから抽出される共通した特徴。「その文字を成り立たせている抽象的な骨組み」を指す。
- **字形** 字体が実物に起こされた際に現れる、具体的な個々の文字の形状。手書き文字、印刷文字を問わず、具体的に出現した個々の文字の形状を指す。
- **文字包摂** 異なる字形を区別せず、それらを同じ文字として捉えること。
- **文字同定** 異なる字形同士が同じ文字かどうかを判断すること。
- **書体** 一定のデザイン上の特徴や様式を備えた文字の集合。
- **手書き書体** 手で書かれた書体。
- **印刷書体** 印刷を行うために生まれ、現代においても印刷や画面表示で利用されている明朝体やゴシック体。
- **構成要素** 1つの漢字の字形において、複数の点画によって構成される一定のまとまりを指す。部首も構成要素に含まれる。
- **部首** 漢字を構成要素に基づいて分類したときに、その基準となる各部の共通する構成要素。康熙字典に定義された214種類の部首を指す。
- **部首番号** 康熙字典の部首の通し番号に基づく1から214までの通し番号。
- **点画** 漢字の字形を構成する最小の要素である点や線。

文字同定作業の予備知識・留意点

文字同定作業を行う際には、まず、その作業の目的と背景を理解することが重要である。この作業の目的は、標準化対象事務に係る現行システムで使用されている文字（以下「使用文字」という）を、行政事務標準文字に同定することである。同定作業を円滑に進めるためには、以下の予備知識を持ち、留意点を理解しておく必要がある。

同定作業者は、以下の点について理解しておく必要がある。

- **使用文字（印刷書体）と手書き書体の関係性についての理解**: 使用文字には、手書き書体から明朝体デザイン化された字形が含まれている。これは、窓口で受け付けた手書き文字を、当時の同定先となる文字集合と同定し、同定できなかった文字を印刷書体の明朝体として再デザインした結果であると推測される。
- **字形から見る手書き書体と印刷書体の違い**: 手書き書体と印刷書体では、デザインの特徴や点画のつながり方に違いがある。例えば、手書き書体では、横画はやや右上がりになりがちだが、印刷書体では水平になる。また、明朝体は、個性を消して、どの職人が描いても同じようになるようにデザインされている。
- **字形を表現する道具による同一文字の字形不一致に関する理解**: 字体が字形となる時の手法は、書体毎に異なる。例えば、アナログであれば石や金属に彫る、紙に筆、万年筆、ボールペンで書く、フィルムで作るといった方法があり、デジタルであればドットを打つ、ベジェ曲線で作るといった方法がある。これらの違いによって、筆運びの痕跡、かすれ、にじみ、データ特有の形などが生まれる。また、同じ字体であっても、人によって書き表そうとしている字形が異なり、細かな違いが生まれる。

包摂可能な部首の字形について

文字の同定作業を行う上で、漢字の構成要素の一つである「部首」の扱いは重要である。漢字を構成する部首は、康熙字典に定義された214種類に分類され、それぞれの部首には特定の字形と画数が定められている。

同定作業においては、この定められた部首の字形と画数を基準として、比較対象となる文字の部首が同一であるかどうかを判断する。ここで重要なのは、**比較対象となる文字間で、部首の位置が同じであれば、部首の字形に多少の違いが見られたとしても、それらを同一の部首として包摂する**ということである。

具体的には、部首の字形の一部に画数の増減や形状の変化が見られる場合でも、それが部首全体の識別を妨げるほど大きな差異でなければ、**部首同士は画数の差異とはせず包摂可能**と判断する。

例えば、部首「辶（しんにょう）」において、点の数が一つ多い、あるいは少ないといった僅かな違いが見られる場合でも、それが「辶」という部首の識別を妨げない限り、同一の部首として扱う。同様に、部首「邑（おおざと）」において、上部の形状が多少異なる場合でも、それが「邑」という部首の識別を妨げない限り、同一の部首として扱う。

このように、部首の字形については、**他の文字構成要素と比較して、より広い範囲で包摂が認められる**。これは、部首が漢字を分類し、検索する上で重要な役割を果たしているためである。部首の字形に厳密な同一性を求めると、本来は同じ部首に分類されるべき漢字が、異なる部首に分類されてしまう可能性が生じる。そのため、部首の識別を妨げない範囲での字形の違いは許容し、同一の部首として包摂することが、効率的かつ正確な文字同定作業に繋がる。

ただし、部首の字形の違いが大きく、**部首の変更を伴うような場合には、包摂の範囲外**となる。例えば、「木」と「禾」のように、明らかに異なる部首を同一の部首として扱うことはできない。

つまり、包摂可能な部首の字形の範囲は、**あくまでも定められた214種類の部首の識別が可能な範囲**であり、その範囲内での字形の微細な差異は許容されるということである。

この原則は、「部首の字形の包摂字例（1)」および「部首の字形の包摂字例（2)」で例示されている通り、個々の部首の字形に適用される。これらの例を参考に、同定作業者は、比較対象となる文字の部首が同一であるかどうかを、柔軟かつ適切に判断することが求められる。

なお、部首の字形に「大小・高低の差」、「長短の差」が含まれる文字について、その文字の部首が変わらない場合は軽微な差として、包摂可能とする。

包摂可能な基準について

一般的に使用される明朝体などの印刷書体においては、同一の文字であっても、デザインの違いにより、字形にわずかな差異が生じることがある。これらの差異は、多くの場合、書体設計上の表現の違い、すなわち**デザイン差**に起因するものであり、**字体の違い**と見なす必要はない。

文字包摂ガイドラインでは、このようなデザイン差を含む字形の差を一定の範囲内で許容し、同一の文字として扱う「**包摂**」の考え方を採用している。つまり、字形のわずかな違いを**字体の違い**と考えるのではなく、**包摂可能な範囲**として定義している。

この包摂の判断基準は、「常用漢字表(平成22年内閣告示第2号)(付)字体についての解説」、総務省「市区町村が使用する外字の実態調査」報告書(2012年3月)、文化審議会国語分科会「常用漢字表の字体・字形に関する指針(報告)」を参考に策定されている。

具体的には、以下に示す基準が、包摂可能な範囲として定義されている。これらの基準は、デザイン差による字形の差を具体的に示すものであり、同定作業において重要な判断指針となる。

1. 大小・高低: 文字を構成する要素の大きさや、上下左右の位置関係（高さ）に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。例えば、「口」の大きさが多少異なる場合や、「木」の縦画の長さが多少異なる場合などが該当する。

2. 長短: 文字を構成する線の長さに多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。例えば、「一」の長さが多少異なる場合や、「木」の枝の長さが多少異なる場合などが該当する。

3. 接触・非接触: 文字を構成する点画同士が、接触しているか、離れているかの違いは、それが文字全体の構成や他の文字との識別に影響を与えない範囲であれば、包摂する。例えば、「口」の右上隅が閉じているか、開いているかの違いなどが該当する。

4. 接触位置: 文字を構成する点画同士が接触する位置に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

5. 交差有無: 文字を構成する点画同士が交差しているか、交差していないかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。例えば、「十」の縦画と横画が交差しているか、接しているだけの違いなどが該当する。

6. 点か線か: 文字を構成する要素が、点として表現されているか、短い線として表現されているかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。例えば、「、」が点か短い線かの違いなどが該当する。

7. 傾斜方向: 文字を構成する線の傾きに多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

8. 曲げ方・折り方: 文字を構成する線の曲がり方や折れ方に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。例えば、「乙」の曲がり具合の違いなどが該当する。

9. 曲げ方・跳ね方: 文字を構成する線の曲げ方と、その先が跳ねているかどうかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

10. 止め・払い: 文字を構成する線の終わりが、止められているか、払われているかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

11. 止め・抜き: 文字を構成する線の終わりが、止められているか、抜かれているかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

12. 止め・跳ね: 文字を構成する線の終わりが、止められているか、跳ねられているかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

13. 八屋根: 文字を構成する要素に「八」の形状が含まれる場合、その形状に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

14. 筆おさえの有無: 文字を構成する線に、筆おさえ（デザイン上の装飾要素）があるかないかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

15. 払い・跳ね: 文字を構成する線の払いの部分が、跳ねているかどうかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

16. 牙の縦横線: 「牙」などの文字に含まれる、牙のような形状をした部分の縦線や横線に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

17. 縦線か横線か: 文字を構成する線が、縦線として表現されているか、短い横線として表現されているかの違いは、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

18. 縦線と左払い: 文字を構成する縦線と、それに続く左払いの形状に多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

19. 運筆方向: 文字を構成する線の運筆方向（書く方向）が、逆になっている場合でも、それが文字全体の構成に影響を与えない範囲であれば、包摂する。

20. 類推判断: 上記の1から19までの基準に明確に当てはまらない場合でも、文字全体を見て、同定対象の文字と類似していると判断できる場合には、包摂する。

これらの基準は、あくまでも例示であり、すべてのデザイン差を網羅しているわけではない。同定作業者は、これらの基準を参考にしながら、個々の文字について、**その違いが文字全体の構成に影響を与え、別の文字との識別を困難にするかどうか**を総合的に判断し、包摂の可否を決定する必要がある。

重要なのは、**字形の違いにとらわれすぎず、文字の本質的な構成要素を見極める**ことである。この「包摂」の考え方を理解し、適切に運用することで、より効率的かつ正確な文字同定作業を実現することができる。

同定先文字の優先設定について

同定支援ツールを用いて同定候補文字を洗い出した結果、複数の候補文字が挙がることがある。その場合は、以下の条件に従って、同定先文字を確定させる必要がある。

複数の同定候補文字が挙がる要因

複数の同定候補文字が挙がる要因としては、主に以下の2つが考えられる。

- **同定先に含まれる同形文字**: 文字情報基盤には、同形異字や戸籍統一文字に重複して収録されている文字が含まれている。そのため、同定先文字に同形文字の組合せが含まれている場合、複数の同定候補文字が挙がることになる。
- **包摂区分の優劣**: 同定支援ツールによる同定結果において、同定候補文字が複数挙がり、その包摂区分の組合せによっては優劣がつけられないことがある。

同定先文字を確定させる条件

複数の同定候補文字が挙がった場合は、以下の優先順位に従って、同定先文字を確定させる。

1. **文字情報基盤文字である**
2. **戸籍統一文字である**
3. **漢字施策（常用漢字）が定義されている**
4. **住基ネット統一文字コードがある**
5. **一意な縮退マップに縮退先が存在する**
6. **実装したUCSがある**
7. **サロゲート範囲ではない**
8. **戸籍統一文字番号の若い文字を優先する**
9. **GJ文字番号が若い文字を優先する**

例えば、同定の優劣が付けられない同定候補文字が2つあった場合、そのうち1つが文字情報基盤文字かつ戸籍統一文字であり、もう一方が文字情報基盤文字ではあるが、戸籍統一文字ではない場合は、前者を同定先として選ぶ。このように、上記の条件を上から順に比較し、優劣がつけられるまで比較を行う。

基底文字の字形への文字同定時の留意点

（本節は文字包摂ガイドラインにはなく本書独自の記載である）自治体による基幹業務システムの標準準拠システムへの移行時には、外字を文字図形番号に同定する。しかしながら他のシステムにおいては、文字図形番号を介さずに、直接UCSコードポイントで管理する場合も考えられる。

その場合に基底文字の字形について、IVSをつけなくとも運用上は同じ字形で表示されることがあるが、運用環境が想定する字形を明確に示すためにIVSをつけることを推奨する。これは将来的に利用される文字フォントの基底文字の字形が変わった場合に、意図しない字形が表示されることを防ぐためである。

JIS2004で例示字形が変更された文字の一部について、IPAmj明朝はJIS90字形を基底文字にしている場合がある。しかしながら今日の情報システムの多くはJIS 2004字形を基底文字としており、周辺のシステムとの相互運用性を考慮して、将来的に基底文字をJIS 2004字形とする可能性が考えられる。そういった仕様の揺らぎに影響を受けないようにするためには、運用環境が想定する字形を表示できる場合であっても、明示的にIVSを指定することが有効と考えられる。

文字コードと自然言語処理

自然言語処理(NLP)における文字コード

本章では、人工知能（AI）技術の利活用において、文字コードが重要な役割を果たすことについて解説する。AIモデルがテキストデータを処理する際、文字コードはデータの解釈に直接影響を与え、その精度や効率に大きな影響を与える。本章では、AI技術、特に自然言語処理（NLP）の分野における文字コードの課題と考慮事項について概説する。

Tokenizerの制限

AI、特に自然言語処理モデルは、テキストデータをトークンと呼ばれる小さな単位に分割して処理する。このトークン化を行うモジュールがTokenizerである。Tokenizerは、単語、文字、サブワードなど、様々な粒度でテキストを分割するが、その設計や実装は特定の文字コード体系に基づいている場合がある。

- **文字コード依存性**: 一部のTokenizerは、特定の文字コード（例えばUTF-8）に最適化されており、異なる文字コードのテキストを適切に処理できない場合がある。
- **未知語 (Unknown Token)**: Tokenizerが学習時に遭遇しなかった文字や文字の組み合わせは、未知語として扱われる。これは、特定の文字コードで表現される稀な文字や、多言語テキストを処理する際に問題となる。
- **サブワード分割**: 近年のTokenizerでは、未知語問題を軽減するためにサブワード分割（例：Byte Pair Encoding (BPE)）が用いられることが多い。しかし、文字コードによっては、サブワード分割の効率が悪くなる場合や、意図しない分割が発生する可能性がある。

データの文字コードに関する注意点

AIモデルの学習データや入力データにおける文字コードの不整合は、様々な問題を引き起こす可能性がある。

- **文字化け**: 学習データと推論時のデータの文字コードが異なる場合、文字化けが発生し、モデルの性能を著しく低下させる。
- **データの前処理**: テキストデータをAIモデルに入力する前に、文字コードを統一する前処理が重要である。一般的にはUTF-8への変換が推奨される。

- **エラーハンドリング**: 文字コードの変換処理においてエラーが発生した場合の適切な処理（例：エラーログの出力、代替文字への置換）を実装する必要がある。
- **データベースとの連携**: データベースに格納されたテキストデータをAIで利用する場合、データベースの文字コード設定を確認し、AIモデルとの間で文字コードの不整合が発生しないように注意する必要がある。

AIにとって理解が難しい文字列

AIモデル、特に初期のモデルや単純な構造のモデルは、特定の種類の文字列を理解することが難しい場合がある。

- **絵文字 (Emoji)**: 絵文字は感情や文脈を豊かにする一方で、文字コードとしては複数のコードポイントで構成される場合があり、Tokenizerやモデルによっては適切に処理できないことがある。
- **記号・特殊文字**: 数学記号、通貨記号、著作権記号など、多様な記号や特殊文字が存在する。これらの文字が学習データに十分に存在しない場合、モデルはそれらの意味を理解することが難しい。
- **制御文字**: 改行コード、タブ文字などの制御文字は、テキストの構造を定義する上で重要であるが、モデルによっては特別な処理が必要となる場合がある。
- **CJK統合漢字以外の漢字**: 日本語、中国語、韓国語で用いられる漢字の中には、CJK統合漢字に含まれない文字（例：CJK互換漢字、CJK補助漢字）が存在する。これらの文字の扱いも考慮が必要となる場合がある。

異体字の取り扱い

異体字とは、同じ意味を持つが字形が異なる文字のことである。Unicodeには、これらの異体字を区別するための仕組みが存在するが、AIモデルの利活用においては注意が必要となる。

- **字形の違いと意味の同一性**: AIモデルは、字形のわずかな違いを意味の違いとして認識してしまう可能性がある。
- **正規化の重要性**: 異体字による影響を軽減するため、テキストデータの正規化（例：NFKC）が有効である。正規化により、異体字は代表的な字形に統一され、モデルは同じ意味を持つテキストをより一貫して処理できるようになる。
- **検索・照合への影響**: AIを用いた検索システムや情報照合システムにおいて、異体字の存在は検索結果の精度に影響を与える可能性がある。正規化を行うことで、より網羅的な検索が可能になる。

日本語テキスト特有の考慮事項

学習用データの正規化

学習用データを正規化することは、AIモデルの性能向上に不可欠である。特に日本語の場合、以下のような正規化処理が重要となる。

- **Unicode正規化**: NFKCやNFCなどのUnicode正規化形式を適用することで、異体字や機種依存文字などを統一的な形式に変換する。例えば、「髙」と「高」、「①」と「1」などを同じ文字として扱うことができるようになる。
- **長音記号の正規化**:「コーヒー」と「珈琲」のように、長音記号の表記揺れを吸収するために、一定のルールに基づいて統一する。
- **半角・全角カナの統一**: 半角カナと全角カナは異なる文字コードを持つため、必要に応じてどちらかに統一する。一般的には全角カナへの統一が推奨される。

LLMは異体字を理解して使いこなせるのか

大規模言語モデル（LLM）は、大量のテキストデータから学習するため、ある程度の異体字や一般的な制御記号（改行、タブなど）を理解し、使いこなす能力を持つ。しかし、初期のモデルや学習データが限られている場合には、異体字を別の文字として認識したり、特殊な制御記号（ゼロ幅スペースなど）を適切に処理できない場合がある。

Transformerベースの LLM（例：GPTシリーズ、BERTなど）は、サブワードトークン化やAttention機構により、文字レベルのわずかな違いや文脈を捉える能力が向上している。そのため、以前のモデルに比べて異体字への対応力は高い。

本稿執筆の過程で、Gemini 2.0 (Gemini Experimental 1206) は、Adobe-Japan 1-6のIVDに基づくIVSを適切に使いこなして、葛飾区と葛城市とを適切に書き分けられている挙動を確認できた。そこから類推するとMoji_Joho IVDに基づくIVSを使いこなせてもおかしくはないが、氏名以外も含めた十分な学習元データが必要と考えられる。

Tokenizerの日本語最適化

日本語は分かち書きをしない言語であるため、Tokenizerは単語の区切りを決定する必要がある。日本語のTokenizerには、以下のような最適化技術が用いられている。

- **形態素解析**: MeCabやJumanなどの形態素解析器を用いてテキストを単語に分割する。品詞情報などを付与することで、より精度の高いトークン化が可能になる。

- **サブワードトークン化**: SentencePiece, BPE (Byte Pair Encoding), WordPieceなどのサブワードトークン化手法は、単語をさらに小さな単位（サブワード）に分割することで、未知語問題への対応力を高める。例えば、「東京タワー」という単語を「東京」「ター」「ワー」のように分割する。これにより、学習データに存在しない単語でも、既存のサブワードの組み合わせとして処理できる可能性が高まる。
- **文字種分割**: 日本語のテキストは、漢字、ひらがな、カタカナ、アルファベットなど、複数の文字種が混在している。Tokenizerによっては、文字種ごとに分割することで、より効率的なトークン化を行う。

これらの技術により、日本語のTokenizerは、単語の曖昧さや多様な表現に対応し、LLMの性能向上に貢献している。Tokenizerの性能は、LLMの性能に直結するため、日本語に特化したTokenizerの研究開発は活発である。評価指標としては、語彙カバレッジ、トークンあたりの平均文字数などが用いられる。

日本語の形態素解析の課題

日本語のテキストをAIで処理する上で、形態素解析は重要な役割を果たす。しかし、日本語の形態素解析には、以下のような課題が存在する。

- **未知語の問題**: 学習データに存在しない単語（新語、専門用語、固有名詞など）を適切に分割できない場合がある。
- **複合語の分割**: 複合語をどこまで細かく分割するかの判断が難しい場合がある。例えば、「情報処理技術者試験」を「情報」「処理」「技術」「者」「試験」と分割するか、「情報処理技術者」「試験」と分割するかなど、文脈によって最適な分割が異なる。
- **同音異義語の判別**: 同じ読み方を持つ異なる意味の単語を、文脈から正しく判別する必要がある。例えば、「橋」と「箸」など。
- **品詞の曖昧性**: 単語の品詞が文脈によって変化する場合がある。例えば、「読む」は動詞だが、「読み」は名詞となる。

これらの課題は、Tokenizerの性能に直接影響し、LLMの言語理解能力を制限する要因となる。最新の研究では、Transformerベースのモデルが、文脈を考慮することで、これらの課題に対してある程度の解決策を示している。形態素解析の誤りは、LLMのダウンストリームタスクの性能に悪影響を与えるため、継続的な改善が求められる。

日本語の文字種とAI

日本語のテキストは、漢字、ひらがな、カタカナ、アルファベット、数字など、多様な文字種で構成される。これらの文字種は、それぞれ異なる役割を持ち、AIの処理において異なる影響を与える可能性がある。

- **文字種の違いと意味**: 一般的に、漢字は意味を、ひらがなは文法的な機能や和語を、カタカナは外来語や強調を表すことが多い。AIモデルは、これらの文字種の違いを学習し、意味理解に役立てることができる。
- **文字種を考慮したTokenizer**: 一部のTokenizerは、文字種ごとにトークンを分割する機能を備えている。これにより、例えば、同じ「computer」でも、ひらがなで書かれた「こんぴゅーた」と区別して処理することが可能になる。
- **文字種変換の重要性**: テキストデータをAIモデルに入力する前に、必要に応じて文字種を変換する（例：半角カタカナを全角カタカナに変換する）ことで、モデルの性能を向上させることができる場合がある。

日本語の表記揺れとAI

日本語のテキストには、同じ意味を持つ単語やフレーズが異なる表記で現れる「表記揺れ」が頻繁に発生する。表記揺れは、AIモデルの学習や推論において、以下のような問題を引き起こす可能性がある。

- **語彙の分散**: 同じ意味を持つ異なる表記が、モデルにとって異なる単語として認識され、語彙が分散してしまう。これにより、モデルは十分な学習データを獲得できず、性能が低下する可能性がある。
- **検索精度の低下**: AIを用いた検索システムにおいて、表記揺れは検索キーワードと一致する文書を見つけにくくする原因となる。
- **データ分析の困難化**: テキストデータ分析において、表記揺れは集計や分析の精度を低下させる。

表記揺れを吸収し、AIモデルの性能を向上させるためには、以下のような対策が考えられる。

- **表記統一**: 学習データや入力データに対して、ルールに基づいた表記統一処理を行う。
- **辞書ベースの正規化**: 表記揺れを吸収するための辞書を作成し、それを用いてテキストを正規化する。
- **類似度学習**: 単語の分散表現などを利用して、意味的に近い異なる表記を同一視するようにモデルを学習させる。

日本語の文体とAI

日本語には、常体（だ・である調）と敬体（です・ます調）、文語体、口語体など、多様な文体が存在する。これらの文体の違いは、AIモデルの学習や推論に影響を与える可能性がある。

- **文体による表現の違い**: 同じ内容でも、文体によって表現が異なるため、モデルは文体を区別して学習する必要がある。
- **文体変換の難しさ**: ある文体で書かれたテキストを別の文体に変換するタスクは、AIにとって高度な処理となる。
- **文体と感情**: 文体は、書き手の意図や感情を伝える役割も果たす。AIモデルが文体を理解することで、より nuanced なテキスト処理が可能になる。

固有表現認識と文字コード

固有表現（人名、地名、組織名など）の認識は、自然言語処理において重要なタスクである。日本語の固有表現は、様々な文字コードで記述される可能性があり、文字コードの違いが認識精度に影響を与えることがある。

- **異体字による表記の多様性**: 同じ人名でも、異体字で表記されることがある。例えば、「斉藤」と「斎藤」など。
- **旧字体・新字体**: 歴史的な文書を扱う場合、旧字体で記述された固有表現を認識する必要がある。
- **文字コード変換の重要性**: 異なる文字コードで記述されたテキストデータを扱う場合、文字コードを統一することで、固有表現認識の精度を向上させることができる。

多言語対応における日本語の特殊性

AIモデルを多言語に対応させる場合、日本語はその文字コードやTokenizerの点でいくつかの特殊性を持つ。

- **文字種の多さ**: 漢字、ひらがな、カタカナという複数の文字種を持つ日本語は、アルファベットのみを使用する言語と比較して、Tokenizerの設計が複雑になる。
- **分かち書きをしない言語**: 日本語は単語間にスペースを入れないため、Tokenizerは形態素解析などの技術を用いて単語を区切る必要がある。これは、英語などの分かち書きをする言語とは異なるアプローチである。
- **異体字や表記揺れの多さ**: 日本語には異体字や表記揺れが多く存在するため、多言語対応を行う際には、これらの日本語特有の課題に対処する必要がある。

- **文字コードの歴史的経緯**: Shift-JISやEUC-JPといった歴史的な文字コードが存在することも、多言語対応における考慮事項となる。

これらの特殊性を考慮し、多言語対応を行う際には、日本語のテキスト処理に特化した技術や知識が必要となる。

AI技術の発展に伴い、文字コードの適切な理解と取り扱いは、モデルの性能向上と安定的な運用に不可欠である。Tokenizerの特性を理解し、Unicodeの利点を最大限に活用しつつ、文字コードに起因する様々な課題に対処することが、AI技術を効果的に利活用するための鍵となる。特に、多言語対応や複雑なテキストデータを扱う場合には、文字コードに関する深い知識と適切な対策が求められる。

文字コードに起因する事故

コンピュータ上で文字を扱う際、異体字や紛らわしい文字、文字セット、文字コード、エンコーディングといった要素が複雑に絡み合い、予期せぬトラブルを引き起こすことがある。本稿では、過去に発生したこれらの要素に起因するインシデント事例をまとめ、その原因や影響、対策について解説する。

異体字・紛らわしい文字に起因する事例

Spotifyにおけるアカウントハイジャック

2013年、音楽ストリーミングサービスSpotifyで、アカウントハイジャックの手法が発見された1。攻撃者は、他人のユーザー名をUnicodeの異体字で登録することで、同じ正規アカウント名に変換される可能性を利用したのだ。例えば、「ᴮᴵᴳᴮᴵᴿᴰ」のような異体字を含むユーザー名を登録すると、「bigbird」という正規のアカウント名に変換されるケースがあった。

Spotifyでは、アプリケーションのレイヤーごとに文字の正規化手法が異なっていたため、この脆弱性を突かれて偽アカウントが登録され、パスワードのリセットが可能となってしまったのだ。

Unicodeでは、同じ文字を複数の異なるコードポイントで表現できる場合がある。例えば、「が」は「か」+「゛」と、「が」1文字で表現できる。このように、同じ文字を異なるコードポイントで表現することを「異体字」と呼ぶ。

Unicode正規化とは、これらの異体字を統一的な表現に変換する処理のことだ。主な正規化形式として、NFCとNFDがある。NFCは合成済み文字を優先する形式で、NFDは分解済み文字を優先する形式である。

Spotifyのケースでは、アプリケーションのレイヤーごとに異なる正規化形式が使用されていたことが問題だった。そのため、あるレイヤーでは異体字として認識された文字が、別のレイヤーでは正規の文字として認識され、アカウントハイジャックが可能となってしまったのだ。

iOSにおけるクラッシュ

2018年、iOSで、特定のUnicode文字を含むメッセージを受信すると、デバイスがクラッシュする脆弱性が発見された1。このメッセージには、white flag、異体字セレクタ、ゼロという3つの文字が含まれていた。

iOSのCoreTextは、正しい異体字を取り出そうとしてパニックを起こし、iOSがクラッシュしたのだ。この問題は、ダイレクトメッセージやグループチャットだけでなく、共有連絡先カードにも影響を及ぼし、iPadや一部のMacbookもこの脆弱性の影響を受けた。

このインシデントは、一見無害に見えるテキストメッセージが、文字エンコーディングの複雑さによって予期せぬ結果を引き起こす可能性を示している。複雑な文字セットと、それを正しく処理できないソフトウェアとの相互作用によって、脆弱性が発生する可能性があることを示す事例である。

紛らわしいドメイン名によるフィッシング

フィッシング詐欺の中でも、ホモグラフ攻撃では、視覚的に似た文字を使用する。例えば、アルファベットの「l」(エル)とアイの大文字「I」、「o」(オー)とギリシア文字の「o」(オミクロン)などを使い分ける。Unicodeの文字セットの拡大は、ホモグラフ攻撃の脅威を著しく増大させている。この問題の本質は、異なる文字体系間で視覚的に区別がつかない文字が多数存在することにある。

文字体系による脆弱性

特に深刻な問題となっているのは、ラテン文字とキリル文字、ギリシャ文字の間に存在する視覚的な類似性である。例えば、ラテン文字の'a'(U+0061)とキリル文字の'a'(U+0430)は、人間の目では区別することが極めて困難である1。

制御文字による新たな脅威

Unicodeには、ゼロ幅スペースや双方向テキストに関する制御文字が含まれており、これらは表示上の違いを生み出さないまま文字列を操作することを可能にする。これらの制御文字は国際化ドメイン名では使用できないものの、他のコンテキストでは悪用の可能性がある。

CJK文字における課題

特に深刻な問題として、CJK互換用文字や異体字の存在がある。例えば「女」という漢字一つを取っても、通常の文字(U+5973)、康熙部首、CJK互換漢字など、複数の異なるコードポイントが存在する。これらの文字は視覚的に区別することが困難であり、フィッシング攻撃に悪用される可能性がある。

ブラウザの対応と限界

現代のブラウザは、複数のスクリプトが混在する場合にPunycodeでの表示を強制するなどの対策を実装している。しかし、単一の文字体系内での類似文字の使用に対しては、有効な対策を講じることが困難である。この問題は、正当な多言語対応とセキュリティのバランスという観点から、完全な解決が難しい課題となっている。

文字コードの設定・符号化に起因する事例

2013年3月に生じた住基ネット障害

2013年に地方自治情報センター（LASDEC）で発生した住民基本台帳ネットワークシステム（住基ネット）のトラブルは、主に文字コード設定の不備に起因するものであった。この障害は、3月26日から27日にかけて、さらに4月1日に再び発生し、全国の231市町村に影響を及ぼした。

3月の障害は、住基ネットのコミュニケーションサーバー（CS）のハードウェアおよびOSを「Windows Server 2008 R2」に更新する際に適用された修正プログラムの不備が原因であった。このプログラムは、住基ネットで使用する特定の文字コードに対応するよう設計されていたが、誤った文字コードでデータベースに情報を保存し、都道府県サーバーに送信するようになった。この結果、都道府県サーバーが本人確認情報の更新処理を停止し、住基ネットの利用が不可能となった。具体的には、住基カードを利用した転入・転出届の処理や広域交付住民票の交付ができなくなるなどの影響が生じた。

この問題は、サーバーの再起動によって一部解消されたものの、都道府県サーバーへの更新情報の送信を回復させるためには、アプリケーションの再修正が必要であった。この修正作業は3月29日から4月2日まで続き、その間、CSの運用が停止されたため、4月1日に再び住基ネットが利用不能となった。これにより、同様の影響が再び発生した。

これらの障害は、住基ネットのシステム更新における文字コード設定の重要性を浮き彫りにした。特に、修正プログラムの設計段階での検証不足が、広範囲にわたるシステム停止を引き起こしたことが問題視される。障害の復旧には、手作業でのデータ修正も含まれ、自治体にとって大きな負担となった。

ファイルアップロードにおけるUnicode正規化問題

ファイルアップロードにおいて、Unicode正規化が原因で問題が発生することがある。特に、Macのファイルシステムでは、NFDという正規化形式が使用されている。

NFDでは、1つの文字が複数のコードポイントに分解されることがある。例えば、「パ」という文字は、「ハ」+「゜」という2つのコードポイントに分解される。

一方、WindowsやLinuxなど、他の多くのOSでは、NFCという正規化形式が使用されている。NFCでは、「パ」は1つのコードポイントで表される。

そのため、Macで作成したファイルをWindowsやLinuxにアップロードすると、ファイル名が正しく認識されないことがある。例えば、「パ」という文字を含むファイル名をアップロードすると、「ハ゜」のように表示されることがあるのだ。

これは、アプリケーションがファイル名を処理する際に、NFCとNFDの正規化形式の違いを考慮していないために起こる。

セキュリティ脆弱性

文字コードやエンコーディングの処理に問題があると、セキュリティ脆弱性につながる可能性がある。例えば、Webアプリケーションで入力値の文字コードチェックが不十分な場合、クロスサイトスクリプティングなどの攻撃に悪用される可能性がある。

クロスサイトスクリプティングとは、Webサイトに悪意のあるスクリプトを埋め込み、ユーザーのブラウザ上で実行させる攻撃だ。攻撃者は、Webアプリケーションの入力フォームなどにスクリプトを埋め込み、ユーザーがそのフォームを送信すると、スクリプトが実行される。

文字コードチェックが不十分な場合、攻撃者は、スクリプトを異なる文字コードでエンコードすることで、チェックを回避することができるのだ。

このようなセキュリティ脆弱性を防ぐためには、入力値の文字コードをチェックし、適切にサニタイズすることが重要である。サニタイズとは、入力値から危険な文字やコードを削除したり、無害な文字に変換したりする処理のことだ。

堅牢な入力値検証と出力エンコーディングは、文字エンコーディングに関連するセキュリティ脆弱性を防ぐために不可欠である。

UTS#39: Unicodeの主なセキュリティ機構

Unicode®は、世界中の文字を単一の文字コード体系で表現することを目的とした国際的な文字符号化規格である。Unicodeは、多言語対応のソフトウェア開発や国際的な情報交換を容易にするために広く利用されている。Unicode® Technical Standard #39の目的は、セキュリティが重要な環境向けに追加の識別子プロファイルを規定することである。Unicodeの広範な文字集合には、セキュリティ上の問題を引き起こす可能性のある文字も含まれている。そのため、Unicode® Technical Standard #39では、Unicodeを使用する際のセキュリティリスク、特に識別子に関する問題を軽減するためのメカニズムが規定されている。

本稿では、Unicode® Technical Standard #39で標準化されたUnicodeの主なセキュリティ機構について解説する。ただし、Unicode® Technical Standard #39では、Unicodeのセキュリティ機構自体は標準化されていないことに注意が必要である。本規格では、セキュリティリスクを軽減するためのガイドラインと推奨事項が提供されている。

識別子におけるセキュリティ

識別子とは、変数名、関数名、クラス名など、プログラム中で使用される名前のことである。Unicodeでは、識別子に利用できる文字を制限することで、セキュリティリスクを軽減している。Unicode Standard Annex #31, "Unicode Identifier and Pattern Syntax" は、どの文字列が識別子として適格であるかを決定するための推奨される方法を提供している。UAX #31の仕様は、文字と数字で識別子を定義するという一般的な慣行をUnicodeのレパートリーに拡張したものである。

具体的には、以下の文字が識別子に使用できない。

文字種	概要
未割り当て文字	まだUnicodeに割り当てられていない文字
私用文字	特定の用途のために予約されている文字
サロゲートペア	Unicodeの拡張領域の文字を表現するために使用される2つの文字の組み合わせ
制御文字（空白文字を除く）	テキストの表示や処理を制御するための文字
非推奨文字	Unicodeで非推奨とされている文字
デフォルトで無視される文字	テキストの表示や処理において無視されるべき文字
NFKC正規化で出現しない文字	NFKC正規化形式で表現できない文字
デフォルトのUnicode識別子として適格でない文字	Unicodeの識別子の定義に合致しない文字
除外スクリプトの文字	セキュリティリスクが高いとされるスクリプトの文字
廃止された文字	もはや現代では使用されていない文字
専門的な用途の文字	技術的な記号や宗教的な記号など、特定の分野でしか使用されない文字
使用頻度の低い文字	現代のテキストではあまり使用されない文字
限定的な用途の文字	使用が限定されているスクリプトの文字

これらの文字を識別子に使用することで、以下のようなセキュリティリスクが発生する可能性がある。

- **なりすまし**: 視覚的に類似した文字を使用して、ユーザーを騙し、悪意のあるWebサイトに誘導する。例えば、ギリシャ文字のΟとラテン文字のOは見た目が非常に似ているため、悪意のあるWebサイトのURLにギリシャ文字のΟを使用することで、ユーザーを騙してアクセスさせることが可能になる。
- **クロスサイトスクリプティング**: 識別子に特殊文字を埋め込むことで、Webページのセキュリティを侵害する。例えば、識別子に <script> タグを埋め込むことで、Webページに任意のJavaScriptコードを実行させることが可能になる。
- **SQLインジェクション**: 識別子にSQL文を埋め込むことで、データベースを不正に操作する。例えば、識別子に DROP TABLE 文を埋め込むことで、データベースのテーブルを削除することが可能になる。

- **パス・トラバーサル**: ファイルシステムへのアクセスを制限するフィルタを回避するために、Unicodeでエンコードされたスラッシュ文字(/)を悪用する。
- **不適切なフィルター回避**: "asshole"のような卑語を禁止するフィルターを回避するために、見た目が似ているUnicode文字を使用する (例: "*sshōle")。

これらのリスクが発生する理由としては、未割り当て文字はプログラムの動作が予測不能になる可能性があり、私用文字は難読化に悪用される可能性があり、制御文字はプログラムの流れを妨害する可能性があるためである 1。

紛らわしい文字の検出

紛らわしい文字とは、視覚的に類似しているが、異なる意味を持つ文字のことである。Unicodeには、多くの紛らわしい文字が含まれており、これらを悪用することで、ユーザーを騙したり、システムを攻撃したりすることが可能になる。

Unicode® Technical Standard #39では、紛らわしい文字を検出するためのメカニズムが規定されている。具体的には、以下の2つのメカニズムが提供されている。

- **スクリプト混在検出**: 異なるスクリプトの文字が混在しているかどうかを検出する。例えば、ラテン文字の "a" とキリル文字の "a" が混在している場合、スクリプト混在検出によって検出される。
- **制限レベル検出**: 文字が特定の制限レベルに準拠しているかどうかを検出する。Unicode® Technical Standard #39では、識別子に使用できる文字を制限レベルによって分類している。制限レベル検出では、識別子に含まれる文字が、指定された制限レベルに準拠しているかどうかを判定する。

これらのメカニズムを使用することで、紛らわしい文字を検出し、セキュリティリスクを軽減することができる。

視覚的ななりすまし

視覚的ななりすましとは、視覚的に類似した文字を使用して、ユーザーを騙す攻撃手法である。Unicodeには、多くの視覚的に類似した文字が含まれており、これらを悪用することで、ユーザーを騙し、悪意のあるWebサイトに誘導したり、偽のソフトウェアをインストールさせたりすることが可能になる。

視覚的ななりすましの例としては、以下のようなものがある。

- ドメイン名におけるなりすまし: 例えば、"apple.com" というドメイン名と、キリル文字の "a" を使用した "apple.com" というドメイン名は、見た目が非常に似ているため、ユーザーを騙して偽の Web サイトに誘導することが可能になる。
- ファイル名におけるなりすまし: 例えば、"report.txt" というファイル名と、ギリシャ文字の "ρ" を使用した "ρeport.txt" というファイル名は、見た目が非常に似ている上に、正規化後には区別ができなくなることから、ユーザーを騙して偽のファイルをダウンロードさせることが可能になる。

視覚的ななりすましを防止するためには、以下の対策を講じることが有効である。

- **ユーザー教育**: ユーザーに対して、視覚的ななりすましの危険性と対策を周知する。
- **ドメイン名登録の制限**: 紛らわしいドメイン名の登録を制限する。
- **ブラウザのセキュリティ機能**: ブラウザに、視覚的ななりすましを検出する機能を実装する。

UTR#36: Unicode におけるセキュリティの考慮事項

Unicode® Technical Standard #39 では、視覚的ななりすましを防止するためのメカニズムは規定されていない。しかし、Unicode® Technical Report #36「Unicode におけるセキュリティの考慮事項」の 2 章では、視覚的ななりすましやその他のセキュリティ問題に関する考慮事項が詳しく解説されている。

視覚的なセキュリティ問題は、主に文字の見た目の類似性を利用した攻撃である。具体的には、以下のようなものが挙げられる。

- 国際化ドメイン名：異なるスクリプトの文字を使用して、正規のドメイン名に酷似したドメイン名を作成する。
- 混在スクリプト・スプーフィング：異なるスクリプトの文字を組み合わせて、正規の文字列に酷似した文字列を作成する。
- 単一スクリプト・スプーフィング：同一スクリプト内で見た目が類似した文字を使用して、正規の文字列に酷似した文字列を作成する。
- 不十分なレンダリングのサポート：一部の文字が正しく表示されない場合に、攻撃者がその文字を悪用して偽の情報を表示する。例えば、文字の形状や位置がわずかに異なる場合でも、レンダリングが不十分な環境では、それらの文字が同一のものとして表示される可能性があり、なりすましのリスクを高める。
  - 悪意のあるレンダリング：意図的に文字を誤ってレンダリングすることで、ユーザーを欺く攻撃。
- 双方向テキスト・スプーフィング：右から左に記述する文字と左から右に記述する文字を組み合わせることで、文字列の表示順序を操作し、偽の情報を表示する。

- 構文スプーフィング：特定の文字の見た目の類似性を利用して、正規の構文に酷似した不正な構文を作成する。
  - 表示されない文字：一部の文字は、特定のフォントや環境では表示されない場合があり、攻撃者はこの特性を利用して、ユーザーに表示されない文字を悪意のあるコードに含める可能性がある。
- 数値スプーフ：見た目が類似した数字を使用して、正規の数値に酷似した偽の数値を作成する。
- IDNA の曖昧性: IDNA (国際化ドメイン名) では、異なる文字列が同じように表示される場合があり、ユーザーが正規のドメイン名と偽のドメイン名を見分けるのが困難になる。

視覚的ななりすましは、一般的なフォントで小さなサイズで通常の画面解像度で表示した場合に、見た目が非常に似ているUnicode文字列を使用することで、人が一方を他方と容易に間違えてしまう状況を突いた攻撃である。視覚的な混同可能性に厳密なルールはない。多くの文字は、十分に小さいサイズで使用すると他の文字のように見える。特に、国際化ドメイン名や様々なスクリプトの利用が増加している現在、視覚的ななりすましは深刻な脅威となっている。

これらの問題を軽減するために、 仕様では以下が要求されている。

- ホスト名の各ラベルは、右から左への文字と左から右への文字の両方を使用してはならない。
- 右から左への文字を使用するラベルは、右から左への文字で開始および終了する必要がある。

制限レベルとアラート

UTR36では、視覚的ななりすましを防ぐために、文字の使用を制限するレベルと、ユーザーに警告を表示するメカニズムが提案されている。

- 制限レベル: 文字の混同可能性に基づいて、識別子に使用できる文字を制限する。
- アラート: ユーザーが偽の識別子にアクセスしようとした場合に、警告を表示する。

視覚的ななりすましを軽減するための手法

視覚的ななりすましを軽減するための手法として、UTR36では、ケースフォールド形式の使用が提案されている。ケースフォールド形式とは、大文字と小文字を区別せずに文字を表現する形式である。

非視覚的なセキュリティ問題

Unicodeには、視覚的ななりすまし以外にも、セキュリティ上の問題を引き起こす可能性のある要素が存在する。Unicode® Technical Report #36では、これらの非視覚的なセキュリティ問題についても解説されている。

非視覚的なセキュリティ問題には、以下のようなものがある。

- **UTF-8の悪用**: UTF-8のエンコーディングの脆弱性を悪用した攻撃。例えば、不正なUTF-8シーケンスをWebアプリケーションに送信することで、バッファオーバーフローを引き起こしたり、クロスサイトスクリプティング攻撃を実行したりすることが可能になる。
- **テキスト比較**: Unicodeの正規化を考慮せずにテキスト比較を行うことによる問題。Unicodeでは、同じ文字を異なるコードポイントで表現することができる。例えば、"é" は、U+00E9 (é) と U+0065 (e) + U+0301 (´) の2つの方法で表現することができる。これらの文字は視覚的には同じであるが、コードポイントが異なるため、正規化を考慮せずに比較を行うと、誤った結果が得られる可能性がある。
- **バッファオーバーフロー**: Unicode文字列のバッファオーバーフロー。Unicode文字列は、ASCII文字列よりも多くのメモリを消費する。そのため、Unicode文字列を扱うプログラムでは、バッファオーバーフローが発生する可能性が高くなる。特に、UTF-8エンコーディングでは、1文字を複数のバイトで表現するため、バッファオーバーフローが発生しやすい。
- **プロパティと文字の安定性**: Unicodeのバージョンアップによる文字プロパティの変更による問題。Unicodeは、定期的にバージョンアップが行われており、その際に文字のプロパティが変更されることがある。そのため、古いバージョンのUnicodeを前提としたプログラムでは、新しいバージョンのUnicodeで動作させた場合、予期せぬ動作をする可能性がある。
- **コードポイントの削除**: セキュリティ上の問題を引き起こす可能性のあるコードポイントの削除。例えば、制御文字を削除することで、プログラムの流れを改変することが可能になる。また、非表示文字を削除することで、ユーザーに重要な情報を見せないようにすることが可能になる。
- **安全なエンコーディング変換**: エンコーディング変換時のセキュリティ問題。エンコーディング変換を行う際に、不正な文字列が生成される可能性がある。例えば、UTF-8からUTF-16に変換する際に、サロゲートペア文字が不正に生成される可能性がある。このような不正な文字列は、プログラムのクラッシュやセキュリティホールの原因となる可能性がある。
- **ロスレス変換**: ロスレス変換を可能にするためのメカニズム。Unicodeでは、異なるエンコーディング間でロスレス変換を行うためのメカニズムが提供されている。ロスレス変換とは、変換の前後で情報が失われない変換のことである。例えば、UTF-8からUTF-16に変換し、再びUTF-8に変換した場合、元のUTF-8文字列と完全に一致する。

- **冪等性**: Unicodeの変換処理の冪等性。冪等性とは、同じ処理を複数回繰り返しても、結果が変わらないという性質のことである。Unicodeの正規化などの変換処理は、冪等性を満たすように設計されている。これにより、変換処理を複数回繰り返しても、結果が変わらないことが保証される 。

UTS #55: Unicode Source Code Handling

Unicodeは、世界中の様々な言語の文字を包含する文字コード体系である。その包括性ゆえに、ソースコードにおいてUnicodeを適切に扱わなければ、ユーザビリティやセキュリティ上の問題が発生する可能性がある。Unicode Technical Standard (UTS) #55 Unicode Source Code Handlingは、これらの問題を軽減するための指針をプログラミング言語設計者およびプログラミング環境開発者に提供することを目的とした文書で、最新版は2024年1月に出版された。Unicode標準への準拠は、UTSへの準拠を必要としないことに注意が必要である。

UTS #55策定の背景と目的

Unicodeは、世界中の多様な文字を単一の文字コード体系で表現することを目指し、多くの文字を収録している。その中には、見た目が似ている文字や、表示方向が異なる文字などが存在する。これらの文字をソースコード中で使用する場合、意図せず誤解を招いたり、セキュリティ上の脆弱性につながったりする可能性がある。

例えば、ギリシャ文字のΑ（アルファ）とラテン文字のAは、見た目が非常に似ている。これらの文字を識別子に用いた場合、人間がソースコードを読む際に誤認し、プログラムの動作に意図しない影響を与える可能性がある。また、右から左へ記述する文字（アラビア文字など）と左から右へ記述する文字（ラテン文字など）が混在するソースコードでは、表示順序が複雑になり、コードの理解を困難にする場合がある。

UTS #55は、このようなUnicodeの特性に起因する問題を解決するために策定された。プログラミング言語の設計者や開発環境の開発者に対して、Unicodeを適切に扱うための指針を提供することで、ソースコードのユーザビリティとセキュリティを向上させることを目的としている。UTS #55は、UTS #39で説明されているようなUnicodeセキュリティメカニズムを補完するものである。UTS #39は、Unicodeにおけるセキュリティ上の問題全般を扱っているのに対し、UTS #55は特にソースコードの取り扱いに焦点を当てている。

UTS #55で解決しようとしている課題

UTS #55は、具体的に以下の課題の解決を目指している。

- **ソースコードの可読性の低下**: 見た目が似ている文字や表示方向が異なる文字の混在により、ソースコードが読みにくくなる問題。
- **セキュリティ上の脆弱性**: 類似文字を利用したなりすましや、表示方向の操作による攻撃の可能性。具体的には、悪意のあるコードを、一見無害なコードのように見せかけることで、開発者を欺き、セキュリティホールを発生させる可能性がある。
- **プログラミング言語間の互換性の問題**: Unicodeの取り扱いがプログラミング言語によって異なる場合に発生する互換性の問題。

これらの課題に対して、UTS #55は、識別子の定義、空白文字や構文の扱い、言語の進化への対応、ソースコードの表示方法、ツールや診断機能など、多岐にわたる指針を提供している。

UTS #55の内容

UTS #55は、以下の3つの主要なセクションから構成されている。

- **プログラミング言語仕様**: 識別子の定義、空白文字や構文の扱い、言語の進化への対応など、プログラミング言語の設計に関する推奨事項を規定している。
- **ソースコードの表示**: ソースコードエディタやレビューツールにおける適切な表示方法を規定している。
- **ツールと診断**: ソースコードの潜在的な問題を検出するためのツールや診断機能に関する指針を提供している。

プログラミング言語仕様

プログラミング言語仕様に関する主要な推奨事項は以下の通りである。

- **正規化と大文字・小文字の扱い**: 識別子にUnicode文字を使用する場合、正規化形式（NFKCなど）を指定し、大文字・小文字の区別を明確にする。これにより、異なるUnicode文字列が同じ識別子として扱われることを防ぎ、プログラムの誤動作やセキュリティ上の問題を回避することができる。
- **空白文字と構文**: Unicodeの空白文字を適切に扱い、構文解析における曖昧さを排除する。Unicodeには、通常のスペース以外にも、様々な空白文字が存在する。これらの文字を正しく処理することで、ソースコードの解釈の誤りを防ぐことができる。
- **言語の進化**: Unicodeのバージョンアップや仕様変更に対応するためのメカニズムを導入する。Unicodeは、新しい文字や機能が追加されるなど、常に進化している。プログラミング言語は、これらの変更に対応できるように設計する必要がある。

ソースコードの表示

ソースコードの表示に関する主要な推奨事項は以下の通りである。

- **双方向性順序**: 右から左へ記述する文字を含むソースコードを正しく表示するためのアルゴリズムを規定している。これにより、表示方向が異なる文字が混在する場合でも、ソースコードを正しく表示し、可読性を確保することができる。
- **空白文字や不可視文字の表示**: 空白文字や不可視文字を視覚的に区別できるように表示する。Unicodeには、目に見えない制御文字や空白文字が多数存在する。これらの文字を特別な記号などで表示することで、ソースコードの構造を理解しやすくする。
- **類似文字の表示**: 類似文字を視覚的に区別できるように表示する。例えば、ギリシャ文字のΑとラテン文字のAを異なるフォントで表示したり、警告を表示したりすることで、誤認を防ぐことができる。
- **構文の強調表示**: Unicode文字を含むソースコードに対しても、構文を正しく強調表示する。これにより、ソースコードの構造を把握しやすくし、可読性を向上させることができる。

ツールと診断

ツールと診断に関する主要な指針は以下の通りである。

- **類似文字の検出**: ソースコード中に類似文字が含まれている場合に警告を表示する。例えば、識別子にギリシャ文字のΑとラテン文字のAが混在している場合、警告を表示することで、開発者に注意を促すことができる。
- **スクリプトの混在の検出**: 異なるスクリプトの文字が混在している場合に警告を表示する。例えば、ラテン文字とキリル文字が混在している識別子は、可読性を低下させる可能性があるため、警告を表示することができる。
- **セキュリティプロファイル**: セキュリティ上のリスクが高い文字の使用を制限する。UTS #39で定義されているセキュリティプロファイルに基づき、なりすましや攻撃に利用される可能性のある文字の使用を制限することで、セキュリティリスクを低減することができる。
- **複数表示形式の検出**: 同じ文字が異なる表示形式で出現する場合に警告を表示する。Unicodeでは、同じ文字でも、異なる表示形式を持つ場合がある。これらの文字が混在すると、可読性を低下させる可能性があるため、警告を表示することができる。
- **ブロックコメントの範囲**: ブロックコメントの開始と終了を明確に識別できるようにする。Unicodeには、様々な種類のコメント記号が存在する。これらの記号を正しく認識し、ブロックコメントの範囲を明確にすることで、コメントの誤解釈を防ぐことができる。
- **方向制御文字**: 方向制御文字の使用方法を検証する。方向制御文字は、文字の表示方向を制御するために使用されるが、誤った使用方法では、ソースコードの表示が乱れる可能性がある。ツールによって方向制御文字の使用方法を検証することで、表示の誤りを防ぐことができる。

UTS #55の主要なポイント、推奨事項、注意点

UTS #55を理解する上で、以下の主要なポイント、推奨事項、注意点を把握しておくことが重要である。

主要なポイント

- Unicodeソースコードの取り扱いは、ユーザビリティとセキュリティの問題を回避するために重要である。
- UTS #55は、プログラミング言語設計者とプログラミング環境開発者向けのガイダンスを提供する。
- UTS #55は、コンピュータ言語を取り巻くツールと仕様のエコシステムの複数のレベルを対象としている。

推奨事項

- 同等の正規化された識別子を使用する。
- 類似文字の軽減診断を実装する。
- ソースコードをその字句構造に従って表示する。
- 方向制御文字の挿入を許可する。
- 不要な方向制御文字を自動的に削除し、正しい方向制御文字を挿入するツールを実装する。

注意点

- ユーザーを混乱させたり、誤解を招いたりする可能性のある文字は多数存在する。
- 方向制御文字を使用する場合は注意が必要である。

本書の制作過程

本書を執筆した目的は行政事務標準文字を理解する上で必要な知識の体系化や、文字コードに関する最新情報の整理、マイナンバー紐付け誤り問題に端を発した氏名・住所などの表記揺らぎについての知見の形式化にある。本来であれば専門家に委ねるべき参考書籍の執筆に、わたしが年末年始に取り組んだ理由としては、これから教科書作成のための調査事業の調達のために数ヶ月かけていては所期の目的に間に合わなかったことと、そういった迅速な整備が求められる文書作成に、AIによってどこまで職員が直接担い得るのか実際に試してみたい思いがあった。

当初は、文字コードという専門的かつ広範な分野の入門書を執筆することは、困難な道のりになるだろうと予想していた。しかし最近のLLMサービスの能力は、期待以上に迅速な執筆を可能にした。本書の文章はわたし自身の経験と学びを記した本章を除いて、例外なくLLMを通じて企画、作成またはレビューされている。とはいえ当初は「おわりに」までGeminiに勝手に書かせてみたところ、私の立場から勝手にあれこれ読者に呼びかけるものだから、さすがに危険を感じて大幅な加筆訂正を行った。

執筆に用いたLLM関連ツール

本書では、基礎的な事実関係の調査にGemini 1.5 Pro with DeepResearchを利用した。当初、Gemini 1.5 Pro with DeepResearchは英語にのみ対応していると認識していたため、調査報告を英語で作成し、Gemini Experimentalで翻訳していた。しかし、年末にDeep Researchが日本語に対応したことに気づき、日本語での調査報告が可能となった。Unicode Technical Standardの解説などは、Deep Researchの出力結果を編集して用いている。

骨格となる構成や文章の作成にはGemini Experimental 1206を利用した。もちろんAIが生成した文章は、時に事実と異なる情報や不適切な表現を含む可能性がある。ハルシネーションが疑われる記載については、Perplexity等の外部Web検索と組み合わせたLLMツールを用いて事実確認を行った。これらのツールを併用することで、LLMの作成した文章をLLMによって事実確認・校閲するフローを構築できた。

Gemini 2.0は直近1〜2年のことに対して、必ずしも適切な回答を返さないケースが見られた。解像度が高いものの古い情報に引っ張られたり、回答に必要な情報をプロンプトに埋め込んだとしても、モデル内部の知識と矛盾する場合に、外部の知識を信じた回答が難しいケースがあった。そこで節ごとに骨格となるストーリーを組み立ててプロンプトに落とし込んだ上で、主に

Perplexityで Claude 3.5を用いて作成し、追記すべき事項を指摘して本文を膨らませる手法を採った。この手法でも一から調べながら手で執筆するのと比べれば、何倍も早く原稿を書き進めることができた。

制作に用いたツール

原稿は全て Visual Studio Code を使って Markdown 形式で作成し、CSS 組版ツールの Vivliostyleでレイアウトを行った。いずれも Webのブラウザエンジンを活用したツールで、問題なく複数フォントを使ってIVSで異体字を書き分けることができ、ここ10年近くでの文字を取り巻くエコシステムの発展を実感できた。

LLMを活用したVSCodeでの文章作成では、ブラウザとエディタ間の頻繁な行き来が発生し、作業効率の向上が課題であった。Github Copilotでは踏み込んだ編集が難しく、途中からVSCode拡張のLLM連携ツールClineを評価した。

当初、原稿は全編を一つのテキストファイルとして編集し、PDFビルド時にPythonスクリプトで分割していた。しかし、Clineへの編集指示において、ファイルサイズが大きいと処理が失敗してしまうケースが多かった。スコープを絞り込みやすいように、分割したファイルをマスターとして編集する方向に切り替え、全編を通しでプロンプトとして流し込む必要がある場合のために、分割した状態で編集した全ファイルを適切な順番で結合するPythonスクリプトを作成した。

GitHub Copilotと比べてもかなり踏み込んで文書編集作業そのものを支援してくれるClineによるModel Context Protocolを使ったエディタとLLMの連携は非常に可能性を感じたが、早々にGeminiのAPI利用制限を使い切ってしまい、誤ってプロジェクトファイルを壊してしまったこと、執筆の後半ではAPI連携できないPerplexityの利用頻度が増えたこともあって、後半ではブラウザとエディタを行き来するスタイルに戻した。Clineに不適切な編集を認めたことで、元のデータを壊してしまう場合もあったことから、慌ててGitHubによる履歴管理を導入した。

LLMを活用してVivliostyleの使い方を習得する過程で、異体字の書き分けに必要なフォント設定のためのCSS組版のカスタマイズに手間取った。その原因は2023年春に大規模刷新する前のブログ記事などをPerplexityが拾ってしまったことにあった。数時間近く嵌まった後に、仕様の刷新を知って2023年春以前の記事を無視する必要に気付いてからは、公式ドキュメントや2023年春以降の記事に絞って読み込んで、そう時間をかけずにキャッチアップできた。

現時点でLLMが得意なこと、苦手なこと

本書の執筆のため10日近く、どっぷりとLLMを使って感じたことは、Geminiでは仕様上のコンテキストウィンドウが広がって、一回のやりとりで出力するテキストの分量が大幅に増えたとはいえ、大雑把な指示で大量の文章を生成させると、依然として多くのハルシネーションが発生する。章構成を考えたり、ストーリーラインを組み立てて、一度に長文を出力してくれるので、本書の骨格を組み立てる過程では重宝したのだが、そのままの文章を使うことは難しく、えらくもっともらしい理屈を組み立てるので、周到に事実確認を行う必要があった。

章の断片ごとにLLMに執筆を指示する際、LLMが内容を深く理解している場合に、他の章で言及されている周辺の話題まで出力してしまう点が悩ましい。この問題に対しては、コンテキストウィンドウに全文を入れて、それらを踏まえた上でこの部分を執筆・修正して欲しいと指示することで改善することを期待したが、コンテキストウィンドウを過度に大きくすると、レスポンスが悪化するだけでなく、適切な文章が生成されないことを確認した。

適切なワークフローを組んでタスクを分割し、それぞれの指示の中で記載内容のスコープを限定する必要があった。インターネット上に情報が存在しないことが明らかな私自身の直近の経験や、LLMに任せても望ましい結果を出しずらい節に限っては、節ごとに書きたいプロットをPerplexityに入力し、事実関係を調べながら節ごとに執筆を薦めるか、自分で文章を執筆した上でGemini 2.0 Flash Thinkingによる査読と表現の見直しを行った。それでも一から調べながら自分で書くよりは、格段に作業効率が高まった。

最後までLLM任せで解説できなかったのは、例えば文字の字体、字形、デザイン差の概念である。字体の説明までは適切にできるのだが、字形、デザイン差に踏み込むと、どうしても書体とか他の説明を始めてしまう。そこで字形、字体、デザイン差についての説明はLLM任せにせず、文字包摂ガイドラインにおける定義をプロンプトとして埋め込む必要があった。

そもそも字体、字形、デザイン差といった概念は異体字に悩まされている漢字圏特有のもので、文字フォントの専門用語として英語圏では字体、字形、デザイン差を日本と同じ文脈では使っていないようである。LLMがどれほど流暢な日本語を操ったとしても、根っこの概念操作は英語の影響を強く受けることを示唆している。

LLMによる異体字のハンドリング

本書において異体字を扱う文章の多くは手作業で修正している。だが一部例外的にIVS/IVDについて紹介した節では、Gemini Experimentalが自発的にAdobe-Japan-1-6 IVDの異体字セレクタを使って書き分けた部分がある。IVSを使って書き分けられたデータが少ない現状で、現在のLLMでIVSを理解して、葛と葛の異体字を書き分けたことは驚きだった。

LLMで異体字を扱うためには、Tokenizerから改良を要するのではないかと心配していたが、技術的には既に対応していることを確認できた。しかしながら他の箇所では異体字を書き分けず、カッコ書きで文字の特徴を記載する場合が多かった。プロンプトでIVSを強く意識した場合のみ、IVSでの異体字を書き分けられたのだろうか。

学習元データとしてよく使われる新聞記事などにおいては、外字の書き分けをカッコ書きの説明としていること、たとえ外字を使っていたとしても、元データにおいてIVSによる書き分けを行っているとは限らないことから、広くIVSによる異体字の書き分けについてデータセットを揃えることは困難と考えられる。これは広く普及しているAdobe-Japan-1-6以上に、住民システムにおける氏名等の表現という特定ドメインに特化したMoji_Joho IVDにとって、深刻な課題となる可能性がある。

LLMによって専門家は不要になるか

本書執筆当初、もしGeminiが容易に書き起こせていたとしたら、それはLLMが学習済みの意味構造を出力しているに過ぎず、教科書を作成する意義はなかったのではないかと懸念した。しかしながらGeminiの幻覚に最初は騙されつつも、ゲラチェックで違和感を感じては一つひとつPerplexityを用いて検証し、周辺を調査して自分自身の文字コードに対する解像度を高めていく過程で、結局のところLLMが抱いている意味体系は混沌とした現実世界の写像に過ぎず、時系列は混乱しているし、真偽も入り交じっていることに気付いた。

歴史的評価が確定した古い過去や規格や議事録の正本が適切に管理されている技術標準に対しては、非常に精度の高い回答を返すことができる一方で、最近の情報を簡単に引くことができないと、つい解像度の高い古い情報に引っ張られてしまう。例えば韓国や北朝鮮の漢字事情について、特に韓国は90年代末からUnicodeに移行し、国内規格としては更新されていないため、あたかも現代に至るまでKS X 1001が使われ続けているような表記となってしまった。より正確な実情を知る上で最も頼りになったのは、京都大学人文科学研究所附属東アジア人文情報学研究センターの安岡孝一先生による2016年の論文だった。北朝鮮に関しても昨今のスマートフォン事情などを見る限り現在はUnicodeが使われている形跡があるところ、信頼できる情報が非常に限られていることから、KPS 9566-2011の記載が厚くなってしまっている。

LLMの進歩は著しく、近い将来、LLMがより適切に時制を認識し、校閲ワークフローを自在に実行するエージェントとして活躍する可能性もある。とはいえ少なくとも現時点においては、職員が体系的に文字コードを把握できる教科書をつくることに意味があり、信頼できる情報を選り分けながら体系的な文書を作成し、外部からの指摘を受けて精度を高めていくことは、LLMの回答精度を高める上でも無駄ではなさそうである。そしてこれは文字コードの話題に限らず、LLM時代のデータマネジメントの在り方を考えていく上でも、ひとつの示唆となりそうだ。

LLMが文書の内容そのものを理解できるようになったとしても、情報の生成日時や作者・有効期限・今なお有効かどうかといったメタデータや、信頼できる情報へのアクセス性の担保は重要であり続けるのだとすれば、AI活用による生産性の向上のためにもデータの適切な管理の重要性は増すとともに、内容の検証や編集といった作業そのものをLLMによって効率化し得ることが、本書の執筆を通じて確認できた。

おわりに

本書「日本語文字コード入門」は、一見すると専門的で難解なテーマを扱っているように見えるかもしれない。しかし、その内容は、私たちが日常的に利用しているスマートフォンやパソコン、そして行政サービスを支える情報システムの根幹に関わるものだ。文字コードは、単なる記号の羅列ではなく、私たちの社会や文化、そしてアイデンティティを形作る重要な要素である。

2023年に発生したマイナンバー紐付け誤り問題を教訓として、デジタル庁は2024年、デジタル社会共通機能グループにデータ品質・標準化チームを新設した。このチームは、行政データの品質向上と標準化を推進する役割を担っている。チームで行政事務標準文字の今後の進め方について検討するにあたり、職員から文字コードについて学ぶための適切な教科書がないかとの相談を受けた。

わたし自身の文字コードとの関わりを振り返ると、1995年ごろ文化庁国語課の職員から「インターネットで日本人は海外の文献にアクセスしやすくなったが、海外の方々が日本語の情報に触れようとしても文字化けして困っている」と相談を受けたことから、文字コードに関心を持つようになった。当時まだ英語版のOSには日本語フォントが組み込まれておらず、海外のPCで日本語を簡単に表示できるのは、独自のフォント管理機構を持ったイスラエル製のブラウザなどに限られていた。

1999年に立ち上がったKondara Projectでは、文字コードに関する様々な課題と論争に直面した。当時まだLinux日本語環境ではEUC_JPが主流である中で、IBMが後にInternational Components for Unicodeと呼ばれる重要なコードベースをオープンソース化し、Linuxの国際化対応が急速に改善される中、KondaraはGNOME、KDEの日本語化やUTF-8への対応、DEC Alphaアーキテクチャーへの対応などに積極的に取り組んでいた。Linuxのエコシステムが急激に商業化しつつもEUCからUTF-8への移行に苦戦する中で、XFree86の分裂とX.orgの再始動、XIMからIIIMFへの移行など、Unicodeとどう向かい合うかは当時の主要な話題の一つだった。

マイクロソフトでは2006年ごろにWindows Vistaの発売準備にあたり、CIO補佐官連絡会議から経済産業省に呼び出され、JIS90字形からJIS 2004字形への変更について大変なお叱りを受けた。率先して最新のJIS規格に対応したことで、JISを所掌する経済産業省でお叱りを受けたことに驚きつつ、わが国の国語施策と字形変更の経緯、文字コードの標準化について丁寧に説明した上で、その後の善後策を協議した。Windows 7からはUnicode IVSに対応し、問題なく「辻」と「辻」を書き分けられるようになった。

多くの霞ヶ関の職員が外字問題を意識したのは、2010年に発足した菅直人政権といわれる。菅直人氏が生まれた1946年は新戸籍法が施行される直前で、当時はまだ任意の漢字で名付けを行うことができ、菅直人氏の「直」の字の縦棒は斜めに傾いているとされた。当時の経済産業大臣の直嶋正行氏にも「直」の字が含まれ、こちらの縦棒はまっすぐと書き分ける必要があった。本省のPCには人名用外字管理が導入されておらず、MS明朝とJS明朝のデザイン差で書き分けて、プリンターフォントを利用しない設定で印刷する必要があったという。菅直人氏の施政方針演説を読むと、着任初日から字形に強くこだわられていたことを確認できる。

同年から立ち上がった文字情報基盤整備事業では、わたしも委員として自治体における過渡期の運用や、変体仮名を日本語文字セットとしてどう扱うべきかなどを議論した。本事業と並行して文字の包摂規準についても整理され、文字を構成する線の傾きに多少の違いがあっても、それが文字全体の構成に影響を与えない範囲であれば包摂することとなった。「直」の縦棒の向きにこだわる政治家は与野党を問わず複数おり、以前は外字として議院のホームページで紹介されていたが、今日では外字ではないと整理されている。閑話休題。

文字コードを体系的に理解できる教科書がないと相談を受けて、私にとっては時代時代の異なる困難に直面して、目の前の課題として取り組んできた文字コードについての諸々が、振り返ると必ずしも体系的には整理されていないことに気付かされた。特にUnicodeが普及して文字化けに悩まされることは減ってからは、文字のビット列を直接扱う低レイヤーの開発者でもない限り、文字コード自体を意識する機会は限られてしまっている。

折しも2023年度から生成AI環境の技術検証環境を構築し、技術文書の作成やレビューに活用していたところ、日本の文字に関する知識や過去の歴史についても、非常に多くの知識を学習していることに気づいた。さらに、Unicodeをはじめとする言語非依存の知識においては、LLMが特に得意な分野であることも確認できた。これらの発見が、LLMを活用した本書の執筆を着想する契機となった。

LLMの助けを借りたことで、本書の草稿は着手からわずか10日ほどで仕上がった。このスピード感は、従来の執筆プロセスでは考えられないものだ。この経験は、生成AIが書籍執筆のあり方を大きく変える可能性を示している。もちろん、AIは万能ではない。特に、文字コードという、長い歴史と複雑な背景を持つ分野においては、専門家の知見が不可欠だ。本書の作成過程では、多くの有識者の方々から貴重なご意見やご指摘をいただいた。査読を通じて、内容の正確性を高め、より実践的な情報を提供することができた。この場を借りて、深く感謝を申し上げる。

2025年は自治体システム標準化の標準準拠システムへの移行にとって極めて重要な年となる。この大規模な移行プロジェクトでは、新旧システム間でのデータ移行、異なるシステム間でデータを円滑にやり取りするために、既存外字の行政事務標準文字への同定が重要な課題となる。さら

に、将来的な基幹業務システムのモダナイゼーションにあたっては、行政事務標準文字を適切に取り扱うために、Unicode IVS/IVDへの対応が不可欠となる。これらの課題に対処する上で、文字コードに関する正確かつ包括的な知識が、これまで以上に重要となる。

本書が、公共情報システムの開発に携わるエンジニアや地方公共団体職員の皆様にとって、日本語文字コードを理解し、適切に扱うための一助となれば幸いだ。そして、文字コードへの理解が深まることで、2025年の自治体システム標準化が円滑に進み、より質の高い行政サービスの提供、ひいては、より豊かで便利な社会の実現に繋がることを心より願っている。

2025年1月 著者

著者: 楠 正憲（くすのき・まさのり）

デジタル庁統括官。デジタル社会共通機能グループ長。

インターネット総合研究所、マイクロソフト、ヤフー、内閣官房、Japan Digital Designなどを経て、デジタル庁の創設に統括官として参画。マイナンバー制度、自治体システム標準化、預貯金２法、ベースレジストリ、トラスト（電子署名法、電子委任状法、Trusted Web）、新技術（AI、Web3）などを担当。

日本語文字コード入門 α版 歴史とUnicodeにおける実装

2025年1月10日 デッカイギ配布版（校閲用、非売品）
2025年5月15日 行政事務標準文字検索 掲載版
2025年5月23日 サイバー犯罪に関する白浜シンポジウム 修正版

発行　楠 正憲

連絡先 MasKusuno@digital.go.jp

注意: 本文書はLLMツール等を用いて作成された文章をベースに、内容の正確性について順次確認と修正を進めているものです。誤りや不正確な記述が含まれている可能性があるため、現時点での記述を過度に信頼せず、必要に応じて一次資料を参照いただくようお願いいたします。誤りを見つけられた場合は、奥付の連絡先までご一報ください。