Chapter 1

Issue

1.1 TEX

1.1.1 The font "Noto Serif CJK SC" cannot be found

在编译文档时提示错误 The font "Noto Serif CJK SC" cannot be found。需要从 GitHub`https://github.com/notofonts/noto-cjk/tree/main/Serif/OTF/SimplifiedChinese`下载对应字体。Noto 是 Google Noto 字体项目，Google Noto 字体项目的目标是为所有现代设备开发一款涵盖所有语言的和谐、优质的字体系列。

Chapter 2

Artificial Intelligence

2.1 ChatGPT

2.1.1 提示词工程最佳实践

虽然我们可以根据自然语言随意的描述需求，但是时刻牢记并遵循如下原则或许有更大机会让 AI 生成更高质量的回答，提升自身与 AI 交互的体验，达到甚至超出我们的预期。本文档的绝大部分内容来源于 OpenAI 的提示词工程官方分享[1]。

使用最新的模型 为了获取更高质量的生成内容，推荐优先使用最新版本的模型。更新的模型可能改进了一些短板，比如降低了生成错误回答的概率，降低了胡说八道的概率。提升了模型的理解能力，提高了模型的响应速度等等。

使用 ### 或者""" 符号来区分指令和文本 例如在需要总结一段文本时，不推荐的做法：

```
请总结下面的文本。
文本内容.....
```

推荐的做法：

```
请总结下面的文本。
文本内容："""
balabala...
"""
```

更具体 尽可能细致的描述您的需求，比如指出长度、格式、风格等等。

[1] https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

通过示例描述你的需求

Chapter 3

TEX

3.1 Font

3.1.1 字体概览

传统西文字体可以粗略的分为衬线字体（Serif)、无衬线字体（Sans-serif)、等宽字体（Monospaced)。衬线字体指的是在字符的末尾添加了小笔画的字体，这些额外的装饰被称为衬线。衬线字体通常用于正文文本，因为衬线有助于提升可读性和流畅性。同时在印刷材料上使用的较多，因为它们有助于长时间阅读。一些常见的衬线字体包括 Times New Roman、Garamond 和 Baskerville。无衬线字体没有在字符末尾添加衬线，因此它们的字母形状更加简洁和直接。无衬线字体通常在数字显示、网页设计和大标题等方面使用，因为它们在屏幕上显示时更容易阅读。常见的无衬线字体有 Arial、Helvetica 和 Verdana。等宽字体中，每个字符都具有相同的固定宽度，无论是宽字符（如"M"）还是窄字符（如"I")。等宽字体主要用于编程和排版需要对齐的文本，以确保字符在垂直方向上对齐。其中一种广为人知的等宽字体是 Courier。现在最标准的分类系统是瓦克斯分类系统（Vox)。最初是由马克西米连·瓦克斯（Maximilien Vox）于 1945 年提出，在 1962 年由国际文字设计协会（the Association Typographique International）采用，2010 年的会议上做了一些细微的调整，将凯尔特体（Gaelic）细分成为一个类别。

1967 年英国标准字体分类(the British Standards Classification of Typefaces)被采用，它基于瓦克斯分类方法，但稍微做了一些简化，在采用后基本上没有变化。

Robert Bringhurst 在《the Elements of Typographic Style》中也根据艺术流派提到了字体的分类，比如：巴洛克风格（Baroque)，洛可可（Rococo)，浪漫主义（Romantic）等风格的字体。

我们主要阐述的瓦克斯分类法 (Vox-ATypI classification)，它是根据字体具有代表性的特定时期（从 15 世纪至今)，基于一些字型上的标准，像：笔画粗细，衬线形式，笔锋的轴线，x 字高等对字体进行区别归类。尽管瓦克斯分类法定义了字体的类别，但是许多字体不仅仅属于一种分类当中。

瓦克斯分类法将字体分为三个大大类：古典风格字体（Classicals)、现代风格字体（Moderns)、书法体（Calligraphics)。再将这三大类细分为人文主义

体（Humanist），加拉德体（Garalde），过渡体（Transitional）——古典风格字体；迪多尼（Didone），机械风格体（Mechanistic），线体（Lineals）——现代风格字体；雕刻体（Glyphic），草体（Script），图形字体（Graphic），黑体（Blackletter）和凯尔特体（Gaelic）——书法字体。其中线体中又细分为了格洛特斯克体（Grotesque），新格洛特斯克体（Neo-grotesque），几何体（Geometric），无衬线人文主义体（Lineal Humanist）。

计算机中渲染的字体可以分为点阵字体 (Dot-matrix-fonts) 和矢量字体 (Vector Fonts)2 大类。在计算机发展早期没有图形界面或图形界面功能简陋之时，字体格式全是点阵字体。所谓点阵字体或更符合其特征的称呼为位图字体 (Bitmap fonts)，每个字形都以一组二维像素信息表示。点阵字体的原理非常简单，对于每一种存在的字号，都存储其特定字符特定字号对应的位图信息，需要时直接输出出来即可。其实在现代 Windows 操作系统中，我们还是可以找到点阵字体的身影。打开一个 cmd，右键标题栏属性，在字体选项卡中我们可以看到默认输出字体就是点阵字体，而且当我们切换字体大小时，我们可以看到字体的样子发生了巨大的改变，看上去根本就不像同一种字体。这恰恰证明了点阵字体是预先设计好直接输出而非即时渲染的特点。对于纯点阵字体，其常见的字体格式包括：bdf，pcf，fnt，hbf[1]。由于图形界面的飞速发展和人们对随心随欲放大缩小字号的需要，设计出来什么字号就只能用什么字号的点阵字体显然并不能满足需求。并且，英文字母是只有 26 个，但是当 CJK 字库加入进来后，维护各种各样字体各种各样大小的中文日文韩文字号显然并不切实际。于是矢量字体或称为轮廓字体 (Outline Fonts) 引入了进来。矢量字体使用贝塞尔曲线描述轮廓，当字体被使用时，计算机即时渲染出对应字号的字体。矢量字体最典型的格式有 PostScript，Type1，TrueType 和 OpenType。TrueType 由 Apple 和微软参与开发，由于 Windows 和 Mac 的普及，是当今使用最多的字体格式。

Type1：全称 PostScript Type1，是 1985 年由 Adobe 公司提出的一套矢量字体标准，使用贝塞尔曲线描述字形，称为 PostScript 曲线。是非开放字体，使用需要收费。

TrueType：TrueType 是 1991 年由苹果（Apple）公司与微软（Microsoft）公司联合提出另一套矢量字标准。虽然与 Type1 都是使用贝塞尔曲线描述字体轮廓，但是 Type1 使用三次贝塞尔曲线来描述字形，而 TrueType 使用的是二次贝塞尔曲线（TrueType 曲线）。TrueType 曲线可接受典型的 hinting，可告知栅格化引擎在栅格化之前应该如何把轮廓扭曲，这样可精确控制字体的抗锯齿结果。

Opentype：是 1995 年由微软（Microsoft）和 Adobe 公司开发的另外一种字体格式，基于 TrueType 扩展，内部兼容了 TrueType 曲线和 PostScript 曲线。并且真正支持 Unicode 的字体，最多可以支持 65535 个码位。其后缀名可以是 ttf 或者 otf。仅包含 TrueType 曲线，其后缀名一般是 ttf，包含有 PostScript

[1]BDF: BDF 是"Glyph Bitmap Distribution Format" 的缩写，也称为 X Window System Bitmap Distribution Format。它是一种用于存储点阵字体的文件格式，最初由 X Window System 使用。

PCF: PCF 是"Portable Compiled Format" 的缩写。PCF 字体格式同样用于存储点阵字体，它是用于 X Window System 的一种改进格式。PCF 格式支持压缩和国际化字符集，并且具有更好的兼容性和可移植性。

FNT: FNT 是"Font" 的简写，用于表示 Windows 系统中的点阵字体文件。FNT 文件包含了字符的位图图像以及字体的相关信息。

HBF: HBF 是"Han Bitmap Font" 的缩写，用于表示汉字点阵字体文件的格式。HBF 格式主要用于存储和处理汉字的位图字形，是针对中文字符的一种点阵字体格式。

曲线的，后缀名则是 otf。

3.2 关于

3.2.1 LaTeX

LaTeX 是一种基于 TeX 的排版引擎。那么 TeX 又是什么呢？TeX 是一个由美国计算机教授高德纳（Donald Ervin Knuth）编写的排版软件。TeX 的 MIME 类型为 application/x-tex，是一款自由软件。它在学术界特别是数学、物理学和计算机科学界十分流行。

20 世纪 60 年代，著名计算机科学家和数学家，斯坦福大学 Donald Knuth 教授在忙于撰写那部叫做《计算机程序设计艺术》的书。这部书计划一共写七卷，Knuth 在写第四卷时，出版社拿来了第二卷的第二版书样给他过目，结果令他大失所望。因为刚刚出现的计算机排版的书籍质量实在无法令 Donald Knuth 教授满意。Donald Knuth 教授的第一版书籍是按照传统的排版方式手工排版，由专业的排版人员实现，手工排版经过几个世纪的发展，能够排版出精美的书籍。但是 19 世纪 60 年代计算机也才出现没多久，C 语言 1972 年才出现，从排版书籍的质量可以了解到，当时排版系统的能力暂时还不能够满足 Donald Knuth 教授的要求。于是 Donald Knuth 教授就计划开发一个符合自己要求的高质量的计算机排版系统。

这个排版系统的名字叫做 TeX。这个名称是由三个大写的希腊字母 T X 组成，在希腊语中是“科学”和“艺术”的意思。为了方便书写，一般在纯文本文档中将其写为 TeX，念做“泰赫”（我个人习惯将其读作“泰克”）。

Knuth 于 1977 年开始构造 TeX 系统，并为该系统设计了一个字体生成软件——METAFONT。1982 年 TeX 系统发布，之后又有几次版本升级。Knuth 用圆周率 的近似值作为 TeX 系统的版本号，并采用自然底数 e 的近似值作为 METAFONT 版本号。系统每升级一次，其版号就增加一位数字，从而不断地趋近于 和 e，这种别出心裁的版本号表示方式一方面可以展示 TeX 与科技文献排版的密切关系，同时也表达了 Knuth 对 TeX 系统与 METAFONT 系统不断追求完美的愿望。1990 年 TeX 第 3.1 版发布时，Knuth 发出宣言：

- 不再对 TeX 进行任何功能上的扩张。
- 如果出现明显问题，则修正后的版本号依次为 3.14，3.141，3.1415..... 在自己离开这个世界的时候，将最后的 TeX 版本序号改为 。此后，即使再发现错误，也都将成为 TeX 的特征而保留。如果有人非要修改的话，就不要再叫 TeX 了，请另外起名。
- 关于 TeX 的一切，已经全部做了书面说明，可以自由利用来设计其他的软件。

TeX 系统的内核相当稳定，几乎没有 bug，1995 年以后版本号一直停止在 3.14159，直到 2002 年 12 月才又进行了一次升级。到目前为止，TeX 系统的版本序号是 3.141592，而 METAFONT 版本序号则为 2.71828。

与时下家喻户晓的微软 Word 那种所见即所得的文档排版方式相比，TeX 显得多么的不合时宜，然而 Knuth 说：『我从来也不期盼 TeX 会成为一个万能的

排版工具，让大家使用它可以来做一些「快速而脏」的东西；我只是将其视为一种只要你足够用心就能得到最好结果的东西。』

Knuth 之所以很自信的宣布不再对 TeX 进行任何功能上的扩张，并非自视甚高，而是有一定客观原因的。其中最主要的原因就是 TeX 提供了宏扩展机制，开发者或者用户均可以对 TeX 提供的 300 多条基本的控制序列（Control Sequence）进行组合，定义更为高级的控制序列，从而增强 TeX 的排版能力。

尽管 TeX 没有原生提供的功能可以通过宏扩展来实现，但是终究是有一些比较重要的问题是宏扩展方式难以解决的，比如对多国语言文字的支持、对 TrueType、OpenType 字体的支持、图形支持等问题。所以在 Knuth 的 TeX 之后，许多人努力地对 TeX 进行改进，或者干脆开发一个全新的“TeX”。开发者们为了尊重 Knuth 宣言的第二条，这些改进版的 TeX 或者重新开发的 TeX 均不再叫 TeX，它们都有新的名字，诸如 e-TeX、Omega、pdfTeX、XeTeX、LuaTeX 等。

自 1990 年以来，TeX 的改进项目层出不穷，但是大浪淘沙，许多项目都没有成功，有的已经死去，有的在苟延残喘。目前广为使用 TeX 改进版本主要有 pdfTeX、XeTeX 和 LuaTeX。应当注意的是 pdfTeX 项目开发者已于 2008 年 6 月宣布自 pdfTeX 1.50.0 版本之后只进行 bug 修正，不再提供新功能扩张。LuaTeX 项目则被看作是 pdfTeX 项目的延伸，并且添加了其它许多重要的新功能，并继续开发下去。

虽然 Knuth 的 TeX 系统出现了许多的改进版本和分化版本，但大都是良性的，并且变动的只是 TeX 引擎，它们大都兼容 Knuth 定义的 TeX 格式。这就好比只改进汽车的发动机性能，并不改变或者只是略微改变汽车驾驶操作方式，这使得用户可以像往常一样去驾驶一辆性能更好的汽车。所以 TeX 用户们通常无需担心 TeX 引擎太多，导致自己的 TeX 文稿在其他人的计算机上无法编译。

Chapter 4

Tricks

4.1 Clean

4.1.1 如何避免电脑养蛊

避免电脑养蛊，最最重要的是要有反蛊意识。从安装的第一个软件做起，要远离流氓，随时警惕，切不可引狼入室，一步错，步步错。一旦安装了流氓软件，其他的流氓软件将会接踵而至。最后的结局是你的电脑会变成流氓的战场。最后的结局可能是 6 个杀毒软件，7 个软件清理工具，8 个浏览器，9 个视频播放器，直到有一天，电脑彻底歇菜。

4.2 GFW

4.2.1 科学上网方式

平时开发中必须要使用 GitHub、StackOverflow 等等，所以流畅的访问他们有助于开发过程中问题的解决。目前比较靠谱的方式是自己搭建服务器或者信誉良好的机场。

Chapter 5

Program

5.1 云原生

5.1.1 减少镜像体积-distroless

由于目前费用紧张，导致服务器的存储资源比较紧张，部署应用时，减少镜像的体积成为了不得不关注的问题。一般构建镜像时，镜像的体积就是基础镜像的体积加应用所占用的体积，在应用体积不变的情况下，减少镜像的体积可以通过选用更小的基础镜像来实现。目前小体积的基础镜像大致如下：

Table 5.1: 基础镜像大小

镜像名称	版本	发布日期	大小
Alpine	3.18	2023-05-29	5MB
Distroless	d70ca864bac5		2.45MB
Busybox	1.36.1	2023-05-19	4.04MB
Scratch	-	-	0MB

Alpine 是一个基于 BusyBox 的轻量级 Linux 发行版。它的特点是非常小巧（通常只有几兆字节），同时提供了完整的包管理系统（APK）。由于其小巧和安全性，Alpine 在容器化应用程序中广泛使用。Alpine 镜像通常用于构建具有基本运行时环境和必要软件包的容器。它可以通过 APK 包管理器来安装其他需要的软件包。Distroless 是谷歌开发的一种极简容器镜像，旨在提供最小化的运行时环境。与其他镜像不同，Distroless 镜像只包含了应用程序运行所需的最小组件，没有操作系统发行版或额外的工具。这使得容器变得更加轻量级、更容易部署和更安全。Distroless 镜像不提供包管理器，通常用于运行无需依赖其他系统库的静态可执行文件或者只依赖少数系统库的应用程序。Busybox 是一个实现了一系列常见 UNIX 工具的单个可执行文件。它的目标是在一个小巧的二进制文件中提供多种命令行工具，如文件操作、文本处理、网络工具等。Busybox 容器镜像通常用于那些依赖许多基本命令行工具的应用程序，它可以提供完整

的 shell 环境和常用的系统工具。Scratch 是 Docker 提供的一个特殊的空白镜像。它只包含了容器运行所需的最小组件，没有任何系统库或工具。Scratch 镜像适合用于构建静态可执行文件或完全不需要操作系统环境的应用程序。由于其极简的特性，Scratch 镜像通常用作最小化和定制化的基础镜像，开发人员可以根据需要自由地向其中添加所需的组件。

以下用实际的示例来说明是如何来减少镜像的体积的，我们的网站后端是用 rust 实现的少量 restful 风格的接口，在没有选用 Distroless 镜像之前，使用的是 debian:bullseye 作为基础镜像，构建完毕后，大概有 70MB+ 的体积，70MB+ 的体积对一个 2023 年的应用来说，确实算不上庞大，但是还是希望能够尽可能的减少体积。根据以往的经验来看，更大的镜像体积，可能会导致更多次生问题。调整为 Distroless 镜像后，Dockerfile 文件如下：

```
ARG BASE_IMAGE=messense/rust-musl-cross:x86_64-musl

# Our first FROM statement declares the build environment.
FROM ${BASE_IMAGE} AS builder

# Add our source code.
WORKDIR /app

COPY . .

RUN rustup target add x86_64-unknown-linux-musl

RUN sudo apt-get update && apt-get install libssl-dev pkg-
        config musl-tools -y

# Build our application.
RUN cargo build --release --target=x86_64-unknown-linux-musl

FROM gcr.io/distroless/static-debian11
LABEL maintainer="jiangtingqiang@gmail.com"
WORKDIR /app
ENV ROCKET_ADDRESS=0.0.0.0
# ENV ROCKET_PORT=11014
#
# only copy the execute file to minimal the image size
# do not copy the release folder
COPY --from=builder /app/target/x86_64-unknown-linux-musl/
        release/alt-server /app/
CMD ["./alt-server"]
```

第一阶段构建出 rust 的可运行程序，第二阶段以最终的可运行二进制程序来构建最小化目标镜像。最终的镜像大小做到了 6MB，只有不到原来大小的 1/10。

5.2 Build

5.2.1 交叉编译和静态链接

静态链接指在编译期间就将所有的依赖库 (包括 Rust 本身库和 C 库等) 都打包到二进制文件中；动态链接则在程序运行时，从操作系统中将所需的库加载到程序中。静态链接的优点是程序一次编译即可在多个三元组相同的目标平台上运行，发布更容易一些，但是库一旦更新就需要重新编译；动态链接则是多个程序共享想同的库，因此程序文件体积更小，且对库更新无感知，但要求目标系统具有该库。对于目标系统类型为 *-*-linux-gnu* 的情形，Rust 一般会将 glibc 和其他库动态链接到编译的二进制文件中。

5.2.2 Rust 静态链接

静态链接库（Statically-linked Library）和动态链接库（Dynamically-linked Library）是两种实现共享函数库概念的实现方式。静态链接指在编译期间就将所有的依赖库 (包括 Rust 本身库和 C 库等) 都打包到二进制文件中；动态链接则在程序运行时，从操作系统中将所需的库加载到程序中。静态链接的优点是程序一次编译即可在多个三元组相同的目标平台上运行，发布更容易一些，但是库一旦更新就需要重新编译；动态链接则是多个程序共享想同的库，因此程序文件体积更小，且对库更新无感知，但要求目标系统具有该库。对于目标系统类型为 *-*-linux-gnu* 的情形，Rust 一般会将 glibc 和其他库动态链接到编译的二进制文件中 1。

为什么要静态链接 目前静态链接最能够解决的痛点就是 Docker 镜像大小，动态链接一般需要依赖 GNU C 标准库，一般情况下采用 GNU C 标准库的基础镜像大小就已经有几十 MB 了。经过初步尝试，静态链接目标镜像初始大小只有 3MB 多，令人惊叹。以最少的资源投入，获取需要的效果。

MUSL 实现静态链接 Rust 实现静态链接的方式之一是使用 MUSL 库，也是最早支持静态链接的方式。默认情况下，Rust 将静态链接所有 Rust 代码。但是，如果使用标准库，它将动态链接到系统的 libc 实现。如果您想要 100%静态二进制文件，可以在 Linux 上使用 MUSL libc。要添加对 MUSL 的支持，需要选择正确的目标。下面以 Linux 平台为例，说明如何实现静态链接。第一步添加 target x86_64-unknown-linux-musl。

```
rustup target add x86_64-unknown-linux-musl
```

Rust 编译时 target 的命名规则为：[arch]-[vendor]-[sys]-[abi]。[arch]：表示目标处理器的架构，如 x86、arm、aarch64 等。[vendor]：可选项，表示提供该平台的组织，如 apple、linux、unknown 等。[sys]：表示目标操作系统或运行时环境，如 linux、windows、android 等。[abi]：可选项，表示二进制接口，即库和应用程序之间的标准化接口，如 gnu、eabi、msvc 等。例如，aarch64-unknown-linux-gnu 中，aarch64 表示系统架构为 ARM 64 位，unknown 表示提供平台的组织未知，linux 表示目标操作系统为 Linux，gnu 表示采用 GNU 的二进制接口。第二步，使用如下的命令构建静态链接的二进制文件：

```
cargo build --target=x86_64-unknown-linux-musl
```

截止 2023 年 06 月，Rust 支持的 Tier 1 级别的 target 组合有：

Target	Notes
aarch64-unknown-linux-gnu	ARM64 Linux (kernel 4.1, glibc 2.17+)
i686-pc-windows-gnu	32-bit MinGW (Windows 7+)
i686-pc-windows-msvc	32-bit MSVC (Windows 7+)
i686-unknown-linux-gnu	32-bit Linux (kernel 3.2+, glibc 2.17+)
x86_64-apple-darwin	64-bit macOS (10.7+, Lion+)
x86_64-pc-windows-gnu	64-bit MinGW (Windows 7+)
x86_64-pc-windows-msvc	64-bit MSVC (Windows 7+)
x86_64-unknown-linux-gnu	64-bit Linux (kernel 3.2+, glibc 2.17+)

Table 5.2: Target Notes

从 2023.07 官方的支持列表可以知道，x86_64-unknown-linux-musl 是 Tier 2 级别的支持仅仅是保证可以 build 成功，不保证运行成功。

C runtime 实现静态链接 从 Rust 1.19 版本开始，支持 C runtime 实现静态链接，不过好像仅仅支持 Windows。从 2020-11-19 发布对 1.48.0 版本开始，开始支持 GNU Linux。构建时传入参数，如下命令所示：

```
RUSTFLAGS='-C target-feature=+crt-static' cargo build --
       release --target x86_64-unknown-linux-gnu
```

5.2.3 Rust 与环境变量

在类 Unix 系统中，如果设置环境变量首先想到的是调整.bashrc/.zshrc。不过在 macOS 上用 Visual Studio Code 开发 rust 应用时，需要注意 Visual Studio Code 启动时无法很好的读取终端的环境变量，至少在终端中导出环境变量后，重启 Visual Studio Code 也是无法读取的，除非从终端中启动 Visual Studio Code，那么 Visual Studio Code 作为终端的一个子进程可以继承终端的环境变量。但需要保证开发过程中需要特定环境变量时必须要从终端中启动 Visual Studio Code，带来不便。一种方式是直接在 Visual Studio Code 启动文件中指定环境变量文件：

```
"envFile": "${workspaceFolder}/.env"
```

然后在环境变量文件中设置应用的环境变量，注意环境变量不需要提交到代码仓库。

5.2.4 Rust 的 Send、Sync Trait

Rust 的野心是提高内存安全性、数据争用安全性（并发）和类型安全性。The stated ambition is improved memory safety, data-race safety (concurrency) and type safety. 而 Send、Sync 就是安全的保证并发的基础 (Send and Sync are fundamental to Rust's concurrency story)。

Send 意味着在单个其他线程上访问 T 是安全的，其中在执行时间线上每一个线程具有一次性的独占访问权。这种类型的值可以通过将独一无二的所有权转移到另一个线程，或者通过独一无二的借用 (&mut T) 在另一个线程上使用。由于在每一个线程上访问 T 的值那一时刻具有独一无二的访问权，从而达到线程安全。此特征的更具描述性的名称可能是 UniqueThreadSafe 。

Sync 意味着多个线程可同时访问 T 是安全的，每个线程都有共享访问权限。这些类型的值可以通过共享引用 (&T) 在其他线程上访问 < 实际上是共享只读而不是写 >。由于共享的是引用不是所有权和借用，共享的是只读，从而达到线程安全。更具描述性的名称是 SharedThreadSafe 。

简单的理解就是 Send 同时只属于一个线程，是独占的。Sync 同一时刻可以在线程之间共享，但是是不可变的。那么既要可变，又要多线程共享，该如何实现呢？主要有 2 种类型来实现，Mutex<T> 和 RwLock<T>, 它们提供了内部可变性, 在内部实现中调用了操作系统的多线程同步机制, 可以保证线程安全。C++、Java 语言中实现线程安全，本质上是使用系统内核或线程库提供的安全 PV 互斥机制运行时来申请一块安全逻辑代码块，在这个逻辑代码块中可以由开发人员加上任意访问 < 读或写 > 对象的代码逻辑，进而实现线程安全访问。1965 年，荷兰学者 Dijkstra 提出了一种卓有成效的实现进程同步和互斥的方法—信号量机制（Semaphore）。信号量其实就是一个变量，我们可以用一个信号量来表示系统中某种资源的数量，比如：系统中只有一台打印机，就可以设置一个初值为 1 的信号量。用户进程可以通过使用操作系统提供的一对原语来对信号量进行操作，从而很方便的实现进程互斥或同步。这一对原语就是 PV 操作：1）P 操作：将信号量值减 1，表示申请占用一个资源。如果结果小于 0，表示已经没有可用资源，则执行 P 操作的进程被阻塞。如果结果大于等于 0，表示现有的资源足够你使用，则执行 P 操作的进程继续执行。可以这么理解，当信号量的值为 2 的时候，表示有 2 个资源可以使用，当信号量的值为 -2 的时候，表示有两个进程正在等待使用这个资源。不看这句话真的无法理解 V 操作，看完顿时如梦初醒。2）V 操作：将信号量值加 1，表示释放一个资源，即使用完资源后归还资源。若加完后信号量的值小于等于 0，表示有某些进程正在等待该资源，由于我们已经释放出一个资源了，因此需要唤醒一个等待使用该资源（就绪态）的进程，使之运行下去。C++、Java 实现的并发容易导致锁的代码块范围过大或其中的对象存在相互依赖导致的互锁等；而 Rust 提供了另一种细粒度 < 区分所有权、是否只读、只在指定类型对象指定代码块加锁 > 的解决方案，它充分利用了语言的静态检查能力，并只在最小的代码块中使用运行时的锁机制来保证线程安全；

5.2.5 Rust 错误处理

主流的错误处理方式

Rust 在设计时，肯定参考了其他语言错误处理的优点和缺点。在了解 Rust 的错误处理方式之前，有必要了解目前所有语言的错误处理主流类型。看看每一

种错误处理的方式有哪些优点和劣势。从而了解为什么要这么设计，在选择时能做出更加合适的决策。

使用返回值（错误码） 使用返回值来表征错误，是最古老也是最实用的一种方式，它的使用范围很广，从函数返回值，到操作系统的系统调用的错误码 errno、进程退出的错误码 retval，甚至 HTTP API 的状态码，都能看到这种方法的身影。

使用异常 因为返回值不利于错误的传播，有诸多限制，Java 等很多语言使用异常来处理错误。

你可以把异常看成一种关注点分离（Separation of Concerns）：错误的产生和错误的处理完全被分隔开，调用者不必关心错误，而被调者也不强求调用者关心错误。

程序中任何可能出错的地方，都可以抛出异常；而异常可以通过栈回溯（stack unwind）被一层层自动传递，直到遇到捕获异常的地方，如果回溯到 main 函数还无人捕获，程序就会崩溃。

使用类型系统

选择合理的错误处理方式

Rust 处理错误时，根据不同的错误采取不同的处理方式，选择最合理的方式来处理。Rust 将错误组合成两个主要类别：可恢复错误（recoverable）和不可恢复错误（unrecoverable）。目前的原则是针对不可恢复的错误，以终止程序为主。可恢复的错误，打印日志为主。没有合适的错误处理原则，会让程序意外结束或者过度的嵌套的模式匹配，影响代码美观。例如获取 Redis 连接时，这样处理：

```
let config_redis_string: String = env::var("REDIS_URL").
       expect("redis url missing");
```

这里获取 Redis 连接串失败时，直接退出应用，因为 Redis 连接串是需要强制存在的配置，无法获取到 Redis 连接串，后续的逻辑也无法完成。直接调用 expect 传入错误信息，结束运行。expect 函数和 unwrap 函数功能点很接近，unwrap 会打印出标准库内置的错误信息，而 expect 函数允许用户定义一个字符串，在程序结束的时候显示。

5.2.6 Rust 生命周期

在 Rust 中每个引用都有自己的生命周期，生命周期保证了引用保持有效的作用域，在大多数情况下这个生命周期是隐式的，是可以被推断出来的，当引用的生命周期可能以不同的方式互相关联时，那就要手动标注生命周期了。

悬垂引用

生命周期最主要的作用就是避免悬垂引用，进而避免程序引用到非预期的数据。例子：

```rust
fn main() {
    let s = String::from("hello").as_str();
    println!("slice: {}", s);
}
```

上面的代码在编译时会提示错误：temporary value dropped while borrowed, consider using a ‘let‘ binding to create a longer lived value。是因为 as_str 只是持有 String 的引用，String 在代码运行结束后会自动 drop 掉，导致 s 变成了悬垂引用。

5.2.7 Rust 内存占用

启动 Rust 应用后，自动被操作系统结束进程，查看操作系统日志。

```
[29569514.446990] Memory cgroup out of memory: Kill process
        26178 (tokio-runtime-w) score 2094 or sacrifice child
[29569514.448350] Killed process 25960 (rss-sync), UID 0,
        total-vm:49068kB, anon-rss:24224kB, file-rss:5316kB,
        shmem-rss:0kB
```

5.3 Performance

5.3.1 优化 React

开启压缩

合理拆包

目前的拆包策略是，尽量保证压缩后的 Chunk 在 100Kb 以内。在 Vite 里是如下的配置：

```
build: {
    outDir: "build",
    rollupOptions: {
        output: {
            manualChunks: {
                react: ['react','react-router-dom','react-dom
        '],
                reddwarf: ['rd-component','rdjs-wheel']
            }
        }
    }
}
```

React 相关的依赖打包成一个 Chunk，公共依赖打包成一个 Chunk，rdjs-wheel 包含所有公共依赖，rd-component 包含公共组件的依赖。

12341223