انواع داده‌ها در زبان‌های Rust و GO

زبان Rust:

انواع داده‌های ساده:

نوع داده	توضیح
i8, i16, i32, i64, i128, isize	عدد صحیح علامت‌دار
u8, u16, u32, u64, u128, usize	عدد صحیح بدون علامت
f32, f64	اعداد اعشاری
bool	مقدار منطقی (true یا false)
char	کاراکتر Unicode ۴ بایتی

انواع داده‌های مرکب:

نوع داده	توضیح
Tuple	مجموعه‌ای از مقادیر با نوع‌های متفاوت
Array	آرایه با اندازه ثابت
Slice	زیرمجموعه متغیر از آرایه
Struct	ساختار سفارشی با فیلدهای مختلف
Enum	نوع شمارشی با قابلیت pattern matching
String	رشته قابل تغییر
&str	رشته غیرقابل تغییر (borrow شده)

زبان GO:

انواع داده های ساده :

نوع داده	توضیح
int, int8, int16, int32, int64	عدد صحیح علامت‌دار
uint, uint8, uint16, uint32, uint64, uintptr	عدد صحیح بدون علامت
float32, float64	عدد اعشاری
complex64, complex128	اعداد مختلط
Bool	مقدار منطقی (true یا false)
Byte	برای داده باینری uint8 معادل
Rune	برای Unicode معادل int32
String	رشته UTF-8

انواع داده‌های مرکب:

نوع داده	توضیح
Array	آرایه با اندازه ثابت
Slice	لیست متغیر و پویا
Struct	ساختار داده‌ای سفارشی
Map	نگاشت کلید به مقدار
Interface	نوع عمومی برای چندریختی
Channel (chan)	برای ارتباط همزمان
Function	توابع به‌عنوان نوع داده

مقایسه بین Rust و GO :

ویژگی	Rust	Go
نوع‌دهی	ایستا و دقیق	ایستا با ساده‌سازی
ایمنی حافظه	GC بسیار بالا، بدون	GC دارای
پیچیدگی	نسبتاً بالا	ساده و قابل فهم
داده مرکب	struct و enum قوی با	map و struct ساده با
رشته‌ها	String و &str با borrow	string غیرقابل تغییر
هم‌زمانی	async/await و tokio با	channel و goroutine با

طراحی انواع داده در Rust و Go: تفاوت‌ها و شباهت‌ها

اهداف طراحی سیستم نوع :طراحی Rust و Go از فلسفه‌های متفاوتی الهام گرفته است Go . زبان ساده‌ای است که برای افزایش بهره‌وری توسعه و مقیاس‌پذیری در سیستم‌های توزیع‌شده طراحی شده است. ساختار نگارش ساده و کم‌کلیدواژهٔ آن بر خوانایی و نگهداری آسان تمرکز دارد. در مقابل، Rustبا هدف ارائه‌ی اطمینان حافظه و کارایی در سطح پایین طراحی شده است . Rustبه‌عنوان یک زبان سیستم‌عاملی که توسط Mozilla Research توسعه یافت، تلاش می‌کند کل دسته‌ای از خطاها به‌ویژه اشکالات مربوط به ایمنی حافظه و همگام‌سازی را در زمان کامپایل از بین ببرد. مدل مالکیت (Ownership) در Rust تضمین می‌کند که مدیریت حافظه در زمان کامپایل بررسی می‌شود و نیازی به وجود زباله‌گیر در زمان اجرا نیست.

سادگی در مقابل ایمنی Go **:**با الگو برداشتن از زبان‌هایی مانند C و با صرفه‌جویی در تعداد ویژگی‌ها و پیچیدگی‌ها، بر سادگی تاکید دارد. این امر باعث می‌شود یادگیری و استفاده از Go سریع‌تر باشد و توسعه‌دهندگان سریعاً بتوانند کدهای مفید بنویسند. در مقابل، Rustزبان غنی‌تر و پیچیده‌تری است که امکانات بیشتری برای ایمن نگه داشتن برنامه‌ها دارد. درRust ، سیستم نوع قوی و مدل مالکیت «ایمنی حافظه» را تضمین می‌کند؛ اما در عوض منحنی یادگیری دشوارتری دارد. به عبارت دیگر، Rustپذیرای محدودیت‌های سخت‌گیرانه‌تری بر توسعه‌دهنده است تا از بروز خطاهای رایج (مثل داده‌های مشترک قابل تغییر یا اشکالات حافظه) جلوگیری کند. بسیاری از تحلیل‌گران معتقدند که Rust اصولا با اِعمال «نظم» بر برنامه‌نویس، باگ‌های زمان اجرا را به حداقل می‌رساند، در حالی که Go دست توسعه‌دهنده را باز می‌گذارد تا در مورد کنترل خطاها و مدیریت منابع سطح انضباط خود را تعیین کند.

مدیریت حافظه و تأثیر آن بر طراحی :مهم‌ترین تفاوت عملی Rust و Go در همین بخش است . Goاز مکانیزم مدیریت حافظه خودکار استفاده می‌کند: حافظه با کمک Garbage Collector در زمان اجرا تخصیص و آزاد می‌شود. این رویکرد ساده‌سازی قابل توجهی برای برنامه‌نویسان به ارمغان می‌آورد (دیگر نیازی به آزادسازی دستی حافظه نیست) و خطر نشت حافظه یا اشاره‌گرهای بی‌ارزش را کاهش می‌دهد. اما این راحتی هزینه‌ای هم دارد: زباله‌گیر در صورت نیاز برنامه را متوقف می‌کند تا حافظه را بازپس گیرد که مقداری سربار زمانی به همراه دارد و در کاربردهایی مثل سیستم‌های تعبیه‌شده یا برنامه‌های Real-Time شاید قابل اغماض نباشد. در مقابل، Rustهیچ زباله‌گیری در سطح زبان ندارد و «مالکیت» و «قرض‌دهی» را جایگزین آن کرده است. هر مقدار (ارزش) دقیقاً یک مالک در زمان اجرا دارد و قواعد دقیقی در زمان کامپایل حکم می‌کند که چگونه این مقدار منتقل یا قرض داده شود. این مدل باعث می‌شود بیشتر مشکلات حافظه (مثل دسترسی از پس یک آزادسازی) پیش از اجرای برنامه شناسایی شوند و برنامه Rust غیرایمن از نظر حافظه اصلاً کامپایل نشود. در نتیجه کد Rust عملکرد بسیار قابل پیش‌بینی و بهینه‌ای دارد، اما توسعه‌دهنده را ملزم می‌کند همواره به مدیریت منابع توجه داشته باشد.

نوع‌دهی ایستا: صراحت و استنباط نوع :هر دو زبان به‌صورت ایستا نوع‌دهی می‌شوند (Static Typing)، اما نحوه استفاده از استنباط نوع (Type Inference) متفاوت است. در Rust ، می‌توان نوع متغیرهای محلی را با `let x = ...` استنباط کرد، اما اغلب نیاز است که نوع پارامترهای توابع و متغیرهای سراسری صراحتاً تعیین شوند. همچنین در Rust ، پیش‌فرض متغیرها تغییرناپذیر (immutable) است و صراحتاً با کلمه کلیدی `mut` علامت‌گذاری می‌شوند. در Go نیز متغیرهای محلی را می‌توان با `:=` تعریف کرد (مثلاً `x := 5` که نوع `x` را خودکار تعیین می‌کند، اما انواع پارامتر تابع یا متغیرهای سراسری را باید با `var a int = 5` یا با تعیین نوع اولیه صریحاً مشخص کرد. هر دو زبان تبدیلات ضمنی (implicit) بین انواع متفاوت را ندارند: مثلاً تبدیل عدد صحیح به اعشاری صریحاً باید انجام شود. به‌طور کلی، Rust در نوع‌دهی سخت‌گیر است و بسیاری از محدودیت‌ها را در زمان کامپایل بررسی می‌کند؛ Go نیز ایستا است اما پیچیدگی کمتری دارد و گاهی پیش‌فرض‌ها و تبیین‌های متنوع‌تری ارائه می‌کند. همانطور که جدول ویژگی‌ها نشان می‌دهد، هر دو زبان «Type: Static typing» هستند.

انواع داده پیچیده و چندریختی :هر دو زبان از انواع داده ترکیبی (Composite Types) مانند ساختارها (Struct) و روش‌های چندریختی پشتیبانی می‌کنند، اما امکانات آن‌ها متفاوت است. در هر دو، تعریف ساختارهای داده با فیلدهای دلخواه ساده است.

Rust:

```
struct Point { x: f64, y: f64 }
let p = Point { x: 0.0, y: 1.0 };
```

Go:

```
type Point struct { x, y float64 }
p := Point{0, 1}
```

برای پشتیبانی از چندریختی، Rust از *Trait* استفاده می‌کند در حالی که Go از *Interface* بهره می‌برد. در Go ، پیاده‌سازی یک واسط (Interface) به‌صورت ضمنی است: اگر یک نوع دارای متدی با نام و امضای مناسب باشد، آن واسط را پیاده‌سازی کرده است. به عنوان نمونه، اگر واسط زیر در Go تعریف شود:

```
type Serializable interface {
    Serialize() []byte
}
```

هر ساختاری که متدی به نام `Serialize()` داشته باشد خودکار پیاده‌سازی `Serializable` محسوب می‌شود. در Rust اما باید صریحاً بنویسیم `impl Trait for Type` تا یک نوع Trait مورد نظر را پیاده کند؛ این صراحت مانع برخی خطاها می‌شود اما انعطاف Go در implicit بودن را ندارد. بنابراین، Go روش آسان‌تری برای

پیاده‌سازی چندریختی ارائه می‌دهد، اما Rust با تحمیل صراحت، امکان بررسی‌های ایمن‌تری فراهم می‌آورد.

علاوه بر این، Rustدارای *Enum*های قدرتمندی است که همان «انواع جمعی جبری » (Algebraic Data Types)هستند و امکان انتساب چندین نوع داده به یک Enum را می‌دهد. برای مثال، در Rust می‌توانیم بنویسیم:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
}
```

سپس با **الگوسازی** (match)می‌توان انواع مختلف Messageرا به‌صورت صریح مدیریت کرد (در بخش بعدی توضیح داده خواهد شد). در Go معادل مستقیم برای این ویژگی وجود ندارد؛ معمولاً با استفاده از interface{}و *type switch*، انواع مختلف را از هم تشخیص می‌دهند. به‌طور مثال در Go می‌توانستیم معادل بالا را چنین بنویسیم:

```
type Message interface{}
type Quit struct{}
type Move struct { x, y int }
type Write struct { text string }

func process(msg Message) {
    switch m := msg.(type) {
    case Quit:
        fmt.Println("Quit")
    case Move:
        fmt.Printf("Move to %d %d\n", m.x, m.y)
    case Write:
        fmt.Println("Text:", m.text)
    default:
        fmt.Println("Unknown")
    }
}
```

اما این شیوه تضمین نمی‌کند که همه حالت‌های ممکن بررسی شده باشند؛ هیچ مکانیزم زبانی اطمینان نمی‌دهد که همه انواع Messageدر زمان کامپایل پوشش داده شوند. مضافاً این که اگر نوع جدیدی به مجموعه پیام‌ها اضافه شود و فراموش کنیم یک شاخه caseدر سوئیچ بنویسیم، برنامه در زمان اجرا کرش می‌کند.

Rustاز سوی دیگر نه تنها *enum*را در زبان تعبیه کرده، بلکه **الگوسازی** (match)را نیز دارد تا بتوان مقادیر پیچیده را به‌راحتی در زمان کامپایل تجزیه و تحلیل کرد. مثلاً برای enum Messageمی‌توان نوشت:

```
match msg {
    Message::Quit => println!("Quit"),
    Message::Move { x, y } => println!("Move to {}
{}", x, y),
    Message::Write(text) => println!("Text: {}",
text),
}
```

که زبان تضمین می‌کند همه حالات Messageپوشش داده شده‌اند (اگر شاخه‌ای حذف شود، اخطار زمان کامپایل صادر می‌شود). در مقابل Go چنین الگوسازی (Pattern Matching) پیشرفته‌ای ندارد و برنامه‌نویس ناچار است از سوئیچ‌های ساده و کنترل‌کننده‌های خطا استفاده کند.

توانمندی‌های چندریختی و ژنریک :هم Rust و هم Go از نوع‌های عمومی (Generics) پشتیبانی می‌کنند، اگرچه Go این قابلیت را از نسخه 1.18 به بعد اضافه کرده است. هر دو زبان در عمل ژنریک‌ها را با تولید نسخه‌ی مخصوص هر نوع (Monomorphization) پیاده می‌کنند. در نتیجه مثلاً اگر یک تابع جنریک را با دو نوع متفاوت فراخوانی کنیم، کامپایلر دو نسخه‌ی مجزا برای آن ایجاد می‌کند. علاوه بر این، روش چندریختی در Rust می‌تواند هم به‌صورت **کامپایل زمان** (Generic + Trait)و هم **زمان اجرا** Trait Object)با &dyn Traitیا (Box<dyn Traitانجام شود. در Go نیز پیش از ژنریک‌ها، فقط **تبادل ژنریک زمان اجرا** با واسطها وجود داشت؛ اکنون با ورود) Generics که از طریق نوع‌های آرایه‌برداری [type parameters] انجام می‌شوند(، پیاده‌سازی مشابه نوع‌کلاس‌ها یا Traits در Rustممکن شده است. با این حال، Goهنوز برخی محدودیت‌های دنیای Trait/Rust را دارد: مثلاً نمی‌تواند به راحتی به «نوع پیاده‌ساز» اشاره کند (یعنی Selfدر سرتاسر امضا) یا تضمین کند که خروجی متد با نوع گیرنده یکی باشد.

الگوسازی و مقایسه داده‌ای Rust :الگوسازی‌سازی (pattern matching) قوی‌ای دارد که با matchو ساختارهای زمانی مانند if letیا while letپیاده می‌شود. به کمک الگوسازی روی enumها وTuple ها، می‌توان کنترل جریان را با وضوح و قطعیت نوشت. به عنوان مثال، در کد قبلی با matchروی Messageدیدیم که تمامی حالات صراحتاً بررسی می‌شوند Go .اما مکانیزم مشابهی ندارد؛ نزدیک‌ترین جایگزین آن «سوئیچ نوع (type switch) «است که فقط برای انواع پویا در رابطها (interface) کار می‌کند. همان‌طور که Frank Moreno اشاره کرده است، در Go به طور ضمنی الگوی الگوسازی وجود ندارد) متاسفانه Go این ویژگی را پشتیبانی نمی‌کند .(در عمل، اگر در Go از switch استفاده کنیم، باید خودمان کنترل کنیم که همه حالت‌های مورد نظر را پوشش دهیم، در غیر این صورت با یک پنیک رو به‌رو می‌شویم. این تفاوت در طراحی باعث می‌شود Rust در کار با داده‌های پیچیده و حالات مختلف وضعیت، ابزار زبان قدرتمندتری داشته باشد.

اثر طراحی انواع داده بر توسعه امن و بدون باگ :طراحی سیستم نوع در هر زبان تأثیر مستقیمی بر کیفیت و امنیت کد دارد Rust .با سیستم نوع قوی و مدل مالکیت خود بسیاری از کلاس‌های خطا (مانند استفاده از حافظه پس از آزادسازی یا دسترسی‌های همزمان ناهمگام) را در زمان کامپایل حذف می‌کند. به‌عنوان مثال، Rustهمانند زبان‌های ایمن حافظه مانندJava ، از کاربر در برابر اشکالات «داده‌ی آزادشده» محافظت می‌کند؛ اما فراتر از آن، در Rust برخلاف Javaو Go امکان وقوع Data Race (دسترسی همزمان چندرشته‌ای خواندن و نوشتن روی یک محل حافظه) در کد ایمن حذف شده است. به بیان دیگر، سیستم نوع Rust امکان دسترسی‌های همزمان ناایمن را پیش از اجرا شناسایی و رد می‌کند. این ویژگی باعث می‌شود برنامه‌های نوشته‌شده با Rust در پروژه‌های بزرگ‌تر، به‌طور قابل‌توجهی امن‌تر و قابل اتکاتر باشند، هرچند پیچیدگی بیشتری را متوجه برنامه‌نویس می‌کند.

Goدر طرف دیگر، اگرچه پایه‌ای ایمن (no manual memory management) دارد، اما نیاز به وِردگونه‌هایی مثل بررسی خطاها و کنترل دستی شرایط خاص برای تولید کدهای بدون باگ بر عهدهٔ توسعه‌دهنده است. همان‌طور که اشاره شد، در Go مدیریت حافظه و همزمانی خودکار است، اما زبان به خودی خود از بروز برخی خطاها جلوگیری نمی‌کند. تحلیل‌ها نشان داده‌اند که «در Go نوشتن یک برنامه مشخص آسان‌تر است، اما احتمالاً نسخه حاصل دارای باگ بیشتری نسبت به نسخه Rust خواهد بود .Rust نظم بیشتری از برنامه‌نویس می‌طلبد، اما در Go خود برنامه‌نویس تعیین می‌کند چه میزان به اصول سخت‌گیرانه پایبند باشد». بدین ترتیب، طراحی Rustسعی دارد با چسباندن قوانین ایمنی به کد، عملیات‌های پرخطر را کاهش دهد، در حالی که Goبا فراهم کردن چارچوب ساده‌تر، بار مسئولیت کنترل باگ‌ها را بیشتر بر دوش برنامه‌نویس می‌اندازد.

نقاط قوت و ضعف Go **:**با سادگی و سهولت نگارش، به سرعت موجب افزایش بهره‌وری می‌شود و برای تیم‌های بزرگ یا برنامه‌های تحت وب مقیاس‌پذیر مفید است، اما به قیمت فداشدن برخی ابزارهای سطح بالا و بررسی‌های کامپایلری پیشرفته در طراحی داده‌ها بوده است Rust .اما با سیستم نوع دقیق و ساختارهای داده‌ی قدرتمند مانند (Enum+Pattern Matching) قابلیت اطمینان بسیار بالاتری فراهم می‌کند، ولی یادگیری آن دشوارتر است و توسعه‌دهنده را مجبور می‌کند به مدیریت حافظه و نوع‌دهی توجه بیشتری کند. در یک جمع‌بندی، انتخاب بین Rust و Goمعمولاً به این بستگی دارد که در پروژه خود «امنیت و کارایی مطلق» را ترجیح می‌دهید یا «سرعت توسعه و سادگی» را. اما در هر حال، فهم عمیق طراحی انواع داده در هر دو زبان برای تولید کدهای امن و قابل نگهداری در پروژه‌های بزرگ حیاتی است.