C Sharp Keys

在 ASP.NET Core 后端开发中，“必须精通”的标准比 Unity 高得多，因为后端要处理高并发、数据库、分布式系统。
以下是后端工程师必须掌握，而 Unity 开发者很少用到的 C# 高级军火库：

1. 异步编程 (Async/Await) —— 后端的命脉

- Unity: 偶尔用 `Task`，大多用协程。
- **后端: 全员异步。**你写的每一行数据库查询、HTTP 请求、文件读写都必须是 `async/await` 的。
- **精通标准:** 懂 `Task`, `ValueTask`, `ConfigureAwait`, 懂线程池饥饿 (Thread Pool Starvation)，知道为什么不能写 `.Result` (防死锁)。

2. 依赖注入 (Dependency Injection / IoC) —— 架构基石

- Unity: 可能用单例 (Singleton) 手搓，或者拖拽赋值。
- **后端: 严禁 `new` 对象。**所有服务都必须通过构造函数注入 (`ISercice provider`)。
- **精通标准:** 理解生命周期： `Transient` (瞬时), `Scoped` (请求级), `Singleton` (全局)。懂得解决循环依赖。

3. LINQ 与 表达式树 (Expression Trees)

- Unity: 为了性能尽量少用 LINQ。
- **后端: 每天都在用。**尤其是配合 Entity Framework (ORM) 操作数据库。
- **精通标准:** 熟练 `GroupJoin`, `SelectMany`。能看懂 `IQueryable` 和 `IEnumerable` 的区别（是在数据库算还是在内存算）。甚至要懂 `Expression<Func<T, bool>>` 这种元编程，用于写动态查询解析器。

4. 反射 (Reflection) 与 特性 (Attribute)

- Unity: 性能杀手，少用。
- **后端: 框架的基础。**你写的 `[HttpPost]`, `[Authorize]` 都是特性。
- **精通标准:** 能自己写一个 AOP (切面) 拦截器，利用反射动态扫描程序集加载插件，或者手写一个简单的序列化工具。

5. 泛型高级用法 (Generics)

- Unity: 会用 `List<T>` 即可。
- **后端:** 需要写通用的 `Repository<T>` 仓储模式。
- **精通标准:** 懂泛型约束 (`where T : class, new()`)，懂协变与逆变 (`out T`, `in T`)，这在设计通用接口时至关重要。

6. 内存优化 (Span / Memory) —— 高性能核心

- Unity: 较少直接操作内存块。
- **后端:** 为了处理高吞吐量的网络包（如解析 JSON 或 TCP 协议），需要使用 **零拷贝** 技术。
- **精通标准:** 熟练使用 `Span<T>`, `Memory<T>`, `ArrayPool`。这是 .NET Core 性能吊打其他语言的秘密武器。

总结

Unity 侧重 **“空间几何与视觉逻辑”** (Quaternion, Vector3, Mesh)。 后端侧重 **“数据流转与并发架构”** (Task, Stream, Expression, DI)。 两者的 C# 技能树点歪了，重合度大概只有 30%。

除了刚才提到的六大核心，ASP.NET Core 后端开发还有几个非常重要、且在面试和实战中必然会遇到的“分水岭”级知识点：

7. 中间件管道 (Middleware Pipeline) —— 请求的生命线

这是 ASP.NET Core 的核心架构模式。

- **概念:** 每一个 HTTP 请求进来，都会穿过一层层“洋葱皮”(中间件)。
- **精通标准:**
  - **执行顺序:** 知道认证 (`UseAuthentication`) 必须在 授权 (`UseAuthorization`) 之前，异常捕获 (`UseExceptionHandler`) 必须在最前面。
  - **手写中间件:** 能写一个 `RequestLoggingMiddleware` 或 `GlobalExceptionMiddleware`，并理解 `next()` 的作用（把请求传给下一棒）。

8. 并发控制与线程安全 (Concurrency & Thread Safety)

后端是多线程环境，成千上万个请求同时在跑。

- Unity: 大部分逻辑在主线程 `Update` 里，很少面对数据竞争。
- **后端:** 必须处理**竞态条件 (Race Conditions)**。
- **精通标准:**
  - **锁:** `lock`, `Monitor`, `SemaphoreSlim` (异步锁)。
  - **原子操作:** `Interlocked.Increment`。
  - **并发集合:** `ConcurrentDictionary`, `ConcurrentQueue` (取代普通的 Dictionary/List)。
  - **ThreadLocal / AsyncLocal:** 用于在同一个请求的上下文中传递数据（比如当前用户信息）。

9. 实体框架 (Entity Framework Core) —— 数据库操作

虽然这是 ORM 库，但它和 C# 语言特性结合极深。

- **精通标准:**
  - **Change Tracker:** 理解 EF 是怎么知道你改了哪个字段的。
  - **N+1 问题:** 知道 `Include()` 预加载，避免循环查库把数据库打挂。
  - **Shadow Properties:** 影子属性（数据库有字段但 C# 类里没有）。

10. 扩展方法 (Extension Methods)

这是让代码变优雅的神器，.NET Core 源码里满地都是这个。

- Unity: 偶尔用来给 `Transform` 加个 `Reset()` 方法。
- **后端: 构建 Fluent API 的基础。**
- **精通标准:** 能通过扩展方法给 `IServiceCollection` 封装模块化的注册逻辑 (`services.AddMyCustomModule()`)。

11. 垃圾回收与非托管资源 (GC & IDisposable)

- **精通标准:**
  - **IDisposable 模式:** 知道 `using` 语句块的本质。如果你写了一个封装了数据库连接或文件流的类，必须正确实现 `Dispose()` 和析构函数，防止内存泄漏。
  - **Finalizer (终结器):** 知道什么时候才需要写 `~ClassName()` (极少情况)。

总结

如果一个 C# 程序员懂 Async/Await + DI + **中间件** + **线程安全**，他就是一个合格的**后端**。 如果他懂 Vector **运算** + **协程** + DrawCall **优化**，他就是一个合格的 Unity **前端**。 两者的世界差别很大。

再挖深一点，到了**架构师/高级开发**级别，还有几个偏向设计模式和底层机制的 C# 领域：

12. 委托的本质与事件总线 (Delegates & Event Bus)

- **基础:** 知道 `Action`, `Func` 怎么用。
- **高级:**
  - **Lambda 闭包 (Closure):** 理解 Lambda 表达式捕获外部变量时的内存陷阱（可能会意外延长对象的生命周期，导致内存泄漏）。
  - **MediatR 模式:** 在后端架构中，经常利用委托和反射实现“进程内消息总线”(CQRS 模式的核心)，将 Controller 和 Service 彻底解耦。

13. 元编程 (Roslyn Quoters / Source Generators) —— 黑魔法

这是 .NET 5 引入的大杀器，现在越来越火。

- **概念:** 代码生成代码。在**编译期间**直接生成 C# 源文件。
- **应用:** 你的 JSON 序列化为什么那么快？你的正则表达式为什么不用解析？因为它们都在编译时被生成了硬编码的代码，而不是运行时反射。
- **掌握:** 能写一个简单的 `IIncrementalGenerator`，自动给所有类生成 `ToString()` 或 Mapper 映射代码。

14. 动态类型 (dynamic / ExpandoObject)

C# 是强类型语言，但它也留了一个动态后门。

- **应用:** 当你需要对接极其不规范的第三方 JSON 数据，或者处理 Python/JavaScript 脚本互操作时。
- **掌握:** 知道 `dynamic` 关键字背后的 `DLR` (动态语言运行时) 机制。知道它有性能损耗，不能滥用。

15. 结构体高级特性 (Struct / Memory Layout)

- **概念:** `class` 是引用类型（堆上），`struct` 是值类型（栈上）。
- **高级:**
  - **StructLayout:** 控制内存对齐（Offset）。在对接 C++ DLL 或读取二进制协议包时，需要精确控制每个字节的位置。
  - **Ref Struct:** 必须分配在栈上的结构体（防止逃逸）。这是 `Span<T>` 的底层原理。
  - **in / ref / out 参数修饰符:** 懂得用 `in` 传递大结构体来避免拷贝，提升性能。

16. 互操作性 (P/Invoke & Marshalling)

后端经常需要调用非托管库（如调用 C++ 写的图像处理算法，或操作系统底层 API）。

- **掌握:**
  - `[DllImport]`: 原生 DLL 调用。
  - **Unsafe 代码 / 指针:** 在 C# 里开启 `unsafe` 块，直接用 `void*` 操作内存指针。虽然危险，但做高性能图像处理/加密时必用。

总结清单

如果你想在简历上写“精通 C# 后端”，你的技能树应该是这样的：

1. **必会:** Async/Await, DI, LINQ, Middleware, EF Core.
2. **拉开差距:** Thread Safety, IDisposable, Memory/Span, Reflection/Attribute.
3. **大神领域:** Source Generators, IL Emit (手写 IL), Unsafe Pointer, GC Tuning.

进阶层 (拉开差距)

5. **线程安全:** `lock`, `SemaphoreSlim`, `ConcurrentDictionary`, `Interlocked`。
6. **内存优化:** `Span<T>`, `Memory<T>`, `ArrayPool` (零拷贝技术)。
7. **反射与特性:** `Reflection`, `Attribute` (编写通用框架基础)。
8. **IDisposable:** 正确管理非托管资源（文件流/数据库连接）。

专家层 (架构师)

9. **委托与事件总线:** Lambda 闭包陷阱，MediatR 模式。
10. **元编程:** **Source Generators** (Roslyn)，在编译时生成代码。
11. **动态特性:** `dynamic` / `ExpandoObject` / DLR。
12. **结构体布局:** `StructLayout`, `Ref Struct`, `in/ref/out` 参数优化。
13. **互操作性:** `P/Invoke`, `Unsafe` 指针操作。
14. **配置与选项模式:** `IOptionsSnapshot`，热重载配置。
15. **可观测性:** `ILogger`, `ActivitySource` (OpenTelemetry 链路追踪)。
16. **性能调优:** GC 调优 (Server GC vs Workstation GC)。