PYTHON ABILITY HANDBOOK

Python 能力强化手册

核心原理、CPython 源码、工程实践、项目映射与面试复习

Python 后端 · MySQL · pytest · Playwright · MCP · 部署与安全

适用方向：Python 后端开发 / 测试开发 / AI 应用与内部工具开发
运行基线：Python 3.11；标注 Python 3.12 - 3.14 的重要差异
版本：2026-08-30

使用说明

这不是一本要求从第一页顺序读到最后一页的教材，而是一套可以反复使用的 Python 能力系统。正文分为三层：

- 速记层：先用一两分钟恢复关键结论。
- 主干层：用十到十五分钟理解原理、运行代码、识别常见错误。
- 深挖层：继续进入 CPython 或框架源码、版本差异和边界条件。

一次复习建议先闭卷回答单元开头的问题，再阅读速记层。已经掌握的单元可以停在主干层；解释不清、项目中出现过问题或面试容易被追问的单元，再进入深挖层。

版本与事实边界

本手册以 Python 3.11 作为代码可运行基线，并核对截至 2026-08-30 的 Python 3.14 稳定系列。CPython 是主要解释对象，但 Python 语言规则不等于 CPython 当前实现。凡是对象内存布局、引用计数、字节码、缓存和 GIL 等实现细节，都会明确标记适用范围。

Flask、pytest、Playwright、MCP 和 FastMCP 等仍会演进。正文解释稳定概念，版本敏感 API 会标注基线和核对入口。项目章节严格区分已经实现的事实、合理推断和可选改进方案。

四级掌握标准

等级	判断标准
能识别	知道概念是什么，能看懂正确示例
能解释	能用自己的话说明机制、失败路径和边界
能使用	能从空白写出最小实现，并通过测试定位问题
能取舍	能比较替代方案，结合项目约束说明选择理由

全书目录：60 个复习单元

目录项可在 PDF 中点击跳转；书签同时提供“篇 - 单元”两级导航。

第一篇：Python 核心与工程基础

第二篇：Python 后端与数据库

第三篇：pytest 与自动化测试

第四篇：MCP 与 AI 应用

第五篇：部署、性能与安全

单元	单元	单元	单元
E01 性能测量与 Profile	E02 内存与资源泄漏	E03 Linux 服务运行	E04 依赖、容器与发布
E05 可观测性	E06 应用安全	E07 故障恢复	

第六篇：项目表达与面试

单元	单元	单元	单元
F01 项目证据	F02 风险分析工具	F03 自动化测试平台	F04 取舍与失败表达
F05 Python 高频题	F06 输出、调试与编码	F07 后端系统设计	F08 模拟面试与掌握法

速查区：全书入口

十分钟复习路径

1. 用一分钟看本周的“模糊 / 不会”列表。
2. 用两分钟闭卷回答一个单元的开场问题。
3. 用五分钟阅读速记层和最小代码。
4. 用两分钟口述项目应用或面试答案。

Python 对象模型速记

问题	结论
变量是什么	名称与对象之间的绑定，不是装数据的盒子
赋值会复制对象吗	通常不会；赋值建立或改变绑定
类型属于谁	类型属于对象，名称可以重新绑定到不同类型对象
`is` 与 `==`	`is` 比身份，`==` 比值语义
可变性是什么	对象在身份不变时，值能否改变
函数如何传参	把实参对象绑定到函数局部形参名称
`del name` 做什么	删除绑定；不保证对象立即销毁

容器与复杂度速记

操作	`list`	`dict` / `set`	选择提示
按下标访问	平均 O(1)	不适用	需要顺序和位置时选序列
按键查找	O(n)	平均 O(1)	需要映射或快速成员判断时选哈希容器
尾部追加	摊销 O(1)	平均 O(1)	列表扩容会发生偶发复制
中间插入	O(n)	不适用	高频队头操作考虑 `deque`
排序	O(n log n)	先转序列	Python 排序稳定；键函数通常只计算一次

并发选型速记

场景	默认起点	主要风险
大量网络 I/O	`asyncio` 或受控线程池	超时、取消、背压、阻塞事件循环

少量阻塞 I/O 库	线程池	线程安全、资源上限、上下文传播
CPU 密集纯 Python	多进程	序列化、进程启动、数据复制
原生扩展释放 GIL	线程可能有效	必须依据库文档和基准测试
Free-threaded CPython	先验证依赖兼容性	线程安全、扩展支持、单线程开销

第一篇：Python 核心与工程基础

A01 Python 程序是怎样运行的

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. `.py` 文件是否会被逐行直接解释执行？
2. 语法错误、导入错误和运行时异常分别可能发生在哪个阶段？
3. `.pyc` 文件是什么，删掉它会影响程序正确性吗？
4. 为什么不能依赖某个 Python 版本的具体字节码编写业务逻辑？

> 速记层
> CPython 通常先把源代码解析为抽象语法树，再编译为代码对象；代码对象包含字节码和运行所需的元数据，最后由解释器执行。模块、函数体和类定义都是代码块，但执行时机不同。字节码属于 CPython 实现细节，不是稳定的跨版本协议。

1. 从源代码到执行结果

把“Python 是解释型语言”理解成“解释器直接逐字逐行读取源文件”并不准确。以 CPython 为例，一段代码通常经历下面的逻辑阶段：

1. 读取与词法处理：识别名称、关键字、数字、缩进等记号。
2. 解析：根据语法规则构建抽象语法树（AST）。
3. 编译：把 AST 编译为代码对象；代码对象中包含字节码、常量、名称表、局部变量信息和源代码位置。
4. 执行：解释器按照当前运行时实现执行代码对象，并创建栈帧、局部命名空间等运行状态。

可以同时观察 AST、代码对象和字节码：

```
import ast
import dis

source = "result = price * count"

tree = ast.parse(source)
code = compile(tree, filename="<demo>", mode="exec")

print(ast.dump(tree, indent=2))
print(code.co_names)
print(code.co_consts)
dis.dis(code)
```

`compile()` 的结果不是最终数值，而是可以被 `exec()` 执行的代码对象。`dis` 展示的是当前 CPython 版本对代码对象的反汇编结果。

> 需要区分三个层次：Python 语言规定代码应产生什么语义；CPython 决定怎样实现这些语义；操作系统和硬件决定进程、线程、内存与机器指令怎样实际运行。

2. 代码块与执行时机

模块、函数体和类定义都是代码块，但定义函数时不会立即执行函数体：

```
print("module start")


def calculate() -> int:
    print("function body")
    return 42


class Job:
    print("class body")


print("module end")
```

导入或执行该模块时，会看到类体中的输出，因为创建类对象需要执行类体代码；函数体只有在调用 `calculate()` 时才执行。

模块顶层代码被执行后，模块对象通常会进入 `sys.modules`。同一解释器进程中再次普通导入时，通常复用缓存中的模块对象，而不是再次执行顶层代码。这个缓存解释了为什么导入副作用、单例状态和测试隔离经常互相影响。

3. `__name__ == "__main__"` 的边界

一个文件作为脚本入口运行时，模块全局名称 `__name__` 通常是 `"__main__"`；作为普通模块导入时，它通常是模块的限定名。

```
def main() -> None:
    print("start service")


if __name__ == "__main__":
    main()
```

这个判断适合隔离命令行入口和导入时行为，但它不是完整的项目架构。服务初始化、配置加载和依赖创建仍应拆分为可测试函数。

> 深挖层：字节码与自适应解释器
> CPython 3.11 引入了更明显的自适应、专门化执行机制。某些通用操作在运行过程中可以被替换或配合缓存，以适应实际遇到的对象类型。`dis.dis(function, adaptive=True, show_caches=True)` 可以观察当前进程中的部分状态，但输出会随执行次数、版本和构建方式变化。它适合排障和学习，不适合作为业务协议或持久化格式。

4. `.pyc` 和导入缓存

导入模块时，CPython 可能把编译结果缓存在 `__pycache__` 下，从而避免下次重复编译。需要记住：

- `.pyc` 主要缓存代码对象，不是源代码加密方案。
- 删除 `.pyc` 通常只会使解释器重新编译，不应改变程序语义。
- 解释器会根据缓存标签和失效信息判断缓存能否复用。
- 不同 Python 版本的字节码不应假设兼容。

- 程序打包、容器镜像和只读文件系统可能改变缓存的写入位置或可用性。

5. 三类错误发生在哪里

错误	典型阶段	示例
`SyntaxError`	解析或编译	括号未闭合、缩进结构非法
`ImportError` / `ModuleNotFoundError`	执行导入语句	模块不存在、导出名称变化
`TypeError` / `KeyError` 等	代码对象执行期间	类型不兼容、键不存在

语法正确只说明代码能被编译，并不说明导入一定成功、分支一定安全或外部依赖一定可用。

> 边界案例
> `compile()`、`eval()` 和 `exec()` 能动态编译或执行字符串，但将不可信输入交给它们等价于给输入者代码执行能力。限制 `globals` 或删除少量内置函数并不能构成可靠沙箱。

6. CPython 源码入口

阅读源码时先找职责，不要一开始追逐每个宏：

- `Python/compile.c`：编译相关流程。
- `Python/ceval.c` 与 `Python/bytecodes.c`：求值循环和字节码定义；具体分工随版本变化。
- `Include/cpython/code.h`：代码对象相关结构和接口。
- `Lib/dis.py`：反汇编工具的 Python 层实现。

源码的 `main` 分支对应开发中的 CPython。需要解释 Python 3.11 项目行为时，应切换到对应维护分支或标签，避免拿新版本布局反推旧版本。

7. 项目应用

在 Flask 或 FastMCP 服务中，如果模块顶层直接读取环境变量、创建数据库连接或启动线程，导入测试模块时也会触发这些副作用。更稳妥的结构是把初始化放入应用工厂或生命周期函数：

```
def create_service(settings):
    repository = build_repository(settings.database_url)
    return RiskService(repository=repository)
```

这样测试可以传入临时配置和替身依赖，导入模块本身不会连接生产资源。

8. 面试追问

问：Python 是编译型还是解释型语言？

建议回答：这个二分法不够精确。Python 语言可以有不同实现；CPython 通常先把源码编译为代码对象和字节码，再由解释器执行。PyPy 等实现可以采用不同执行策略，因此不应把 CPython 当前字节码当成 Python 语言定义。

问：**`.pyc`** 能保护源码吗？

不能。它主要是编译缓存，包含可反汇编的代码对象信息，不应被视为安全边界。

> 练习
>
> 1. 分别对模块顶层代码和函数执行 `dis.dis()`，比较 `STORE_NAME` 与局部变量相关指令。
> 2. 创建一个包含导入副作用的模块，连续导入两次，再删除 `sys.modules` 中的条目观察差异。
> 3. 用 `compile()` 分别以 `exec`、`eval` 和 `single` 模式编译代码，解释适用场景。

掌握标准

- 能画出源码、AST、代码对象、字节码和执行之间的关系。
- 能区分语言规则、CPython 实现和版本细节。
- 能解释模块顶层代码的执行时机及导入副作用。
- 能使用 `ast`、`compile`、`dis` 和 Traceback 定位问题。

官方参考：Python Execution model；dis；CPython source

A02 变量、对象、引用与类型

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 执行 `a = [1, 2]` 时，是把列表装进变量 `a` 吗？
2. 执行 `b = a` 时，会复制一个新列表吗？
3. 一个名称有没有永久固定的类型？
4. `del a` 是删除名称，还是立即销毁对象？
5. 为什么 `id()` 的数值不能当作跨进程或持久化标识？

> 速记层
> Python 数据模型中的数据由对象表示。变量更准确地说是名称，赋值让名称绑定到对象。对象具有身份、类型和值；类型属于对象，不属于名称。赋值通常不复制对象，重新赋值通常只改变绑定。

1. 五个容易混淆的概念

概念	准确定义
对象	Python 程序中的数据实体
名称	用来访问对象的标识符，例如 `rows`、`user`
绑定	名称、属性或容器位置与对象建立关联
身份	对象生命周期内的唯一标识，可通过 `id()` 观察
类型和值	类型决定支持的操作；值是对象表示的数据状态

下面这行代码可以理解为先创建列表对象，再让名称 `a` 绑定到它：

```
a = [1, 2]
```

继续赋值不会自动复制列表：

```
a = [1, 2]
b = a

assert a is b
```

此时 a 和 b 是两个名称，但访问同一个列表对象。

2. 原地修改与重新绑定

```
a = [1, 2]
b = a

b.append(3)
assert a == [1, 2, 3]
assert a is b

b = [9]
assert a == [1, 2, 3]
assert b == [9]
assert a is not b
```

append() 修改的是已有列表对象；b = [9] 创建新列表并重新绑定名称 b。不要把前一种现象解释成“b 的修改同步到了 a”，因为根本不存在两个需要同步的列表。

执行步骤	a 的绑定	b 的绑定	对象状态
a = [1, 2]	L1	未定义	L1 为 [1, 2]
b = a	L1	L1	两个名称共享 L1
b.append(3)	L1	L1	L1 变为 [1, 2, 3]
b = [9]	L1	L2	L1 不变，新建 L2

3. 类型属于对象

```
value = 10
print(type(value))

value = "ten"
print(type(value))
```

名称 value 先后绑定到整数对象和字符串对象。更准确的说法不是“变量改变了类型”，而是“名称重新绑定到了另一种类型的对象”。

Python 通常被描述为动态类型语言：对象类型和很多类型检查发生在运行时，名称可以绑定不同类型对象。Python 也通常被归类为强类型语言，因为它不会随意把不兼容对象偷偷转换后运算：

```
10 + "20"  # TypeError
```

类型注解表达静态预期，不会改变名称绑定模型：

```
count: int = 10
```

```
count = "ten"  # 运行时默认允许；静态检查器应报告问题
```

4. 从字节码看“绑定”

在模块代码中观察赋值：

```
import dis

code = compile("a = 42", "<demo>", "exec")
dis.dis(code)
```

当前 CPython 通常先加载常量，再执行与名称存储相关的字节码。函数局部名称可能使用不同的快速局部变量指令。具体指令名和布局会随版本调整，但核心语义仍是：计算右侧表达式得到对象，再把赋值目标绑定到该对象。

> 深挖层：**`PyObject`** 只解释 CPython
> 在传统 CPython 构建中，许多对象都可以通过以 `PyObject` 头部开头的 C 结构访问。这个头部至少承载类型关联，并在常见构建中与引用计数机制相关。变长对象还需要长度信息。Free-threaded 构建、immortal objects 和不同版本会改变字段、位宽或管理策略，因此面试中可以解释概念，不能把某一版结构体逐字背成永久 ABI。

概念化示意，不代表所有版本的精确源码：

```
typedef struct {
    /* reference-management state: implementation-specific */
    PyTypeObject *ob_type;
} PyObject;
```

阅读当前源码应从 `Include/object.h` 开始，再跟到具体类型实现，例如 `Objects/listobject.c`。如果讨论 Python 3.11 行为，应查看 3.11 对应标签，而不是只看 `main`。

5. `id()`、`is` 与实现细节

语言层保证对象在生命周期内有唯一身份，`id(obj)` 返回能代表该身份的整数。CPython 中它经常与内存地址有关，但代码不应依赖这一点：

- 对象释放后，新的对象可能复用同一个 `id` 数值。
- 不同进程中的 `id` 没有可比较意义。
- 其他 Python 实现可以采用不同策略。
- 小整数、字符串常量可能被缓存或驻留，导致某些 `is` 结果看似稳定。

因此值比较使用 `==`，单例身份判断使用 `is`，最常见的是：

```
if result is None:
    ...
```

> 边界案例：不要用常量缓存猜语义
> 同样写出两个值相等的整数或字符串，`a is b` 可能因解释器、代码编译方式和运行上下文不同而变化。业务代码只能依赖 `a == b` 的值语义。

6. `del` 删除绑定，不承诺立即销毁

```
items = [1, 2]
backup = items

del items
print(backup)  # [1, 2]
```

`del items` 删除名称 `items` 的绑定。只要还有可达引用，对象仍然存在。即使最后一个明显引用被删除，Python 语言也不承诺某个跨实现一致的立即析构时刻。CPython 的引用计数常使资源看似立即释放，但文件、锁、数据库连接等资源仍应通过上下文管理器显式管理。

7. 项目中的真实风险

提审服风险排查工具取得原始查询结果后，如果直接共享列表再排序：

```
raw_rows = query_database()
report_rows = raw_rows

report_rows.sort(
    key=lambda row: row["login_count"],
    reverse=True,
)
```

`raw_rows` 的顺序也会改变。如果后续 HHI 或分位数逻辑错误地依赖原顺序，问题会很隐蔽。只需要独立顺序时可以使用：

```
report_rows = sorted(
    raw_rows,
    key=lambda row: row["login_count"],
    reverse=True,
)
```

不过两个列表内部的字典仍然共享，这属于浅拷贝边界，将在 A04 继续处理。

8. 面试追问

问：Python 变量保存的是什么？

建议回答：更准确地说，Python 使用名称绑定对象。对象具有身份、类型和值；赋值让名称绑定到右侧表达式产生的对象，通常不会复制对象。

问：Python 是动态类型还是弱类型？

建议回答：Python 是动态类型语言，对象类型和很多检查发生在运行时；它通常也被归为强类型语言，不兼容类型一般不会被隐式转换后直接运算。“强弱类型”本身不是严格统一的语言规范术语，所以要说明所指行为。

问：**`del x`** 会释放内存吗？

建议回答：它先删除绑定。如果对象仍被引用，就不会消失；即使没有引用，释放时机也涉及实现和垃圾回收，不能把 `del` 当作可靠的外部资源释放机制。

> 练习
>
> 1. 预测 `a = [1]; b = a; a = a + [2]` 后 `a` 与 `b` 的值和身份关系。

> 2. 解释为什么 `primary = backup = []` 容易造成共享状态。
> 3. 分别在模块顶层和函数内部反汇编一次赋值，说明指令差异属于什么层次。
> 4. 设计一个测试，证明 `sorted(rows)` 与 `rows.sort()` 对原列表的影响不同。

掌握标准

- 能准确使用名称、绑定、对象、身份、类型和值这六个术语。
- 能预测共享可变对象、原地修改和重新绑定的结果。
- 能解释 `id()` 与 `is` 的语言保证和 CPython 实现边界。
- 能在项目中发现共享状态，并选择重新构造、浅拷贝或深拷贝。

官方参考：Python Data model；CPython Include/object.h

A03 可变对象、不可变对象与参数传递

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. “可变”描述的是名称还是对象？
2. 为什么函数修改列表后，调用者能看到变化？
3. Python 是值传递还是引用传递？怎样回答才不含糊？
4. 为什么可变默认参数会跨多次调用保留状态？

> 速记层
> 可变性描述对象在身份不变时，值能否改变。函数调用会把实参对象绑定到新的局部形参名称；函数既可能修改共享可变对象，也可能只重新绑定局部名称。默认参数在函数定义时求值一次，而不是每次调用重新创建。

1. 可变性是对象的性质

列表、字典和集合通常是可变对象；整数、字符串、字节串和元组通常是不可变对象。不可变不等于“名称不能换对象”：

```
count = 1
count = count + 1
```

这里没有把整数对象 `1` 改成 `2`，而是计算出新的整数对象，再重新绑定 `count`。列表的原地修改则保持对象身份：

```
items = [1]
before = id(items)
items.append(2)
assert id(items) == before
```

元组自身不可变，但可以包含可变对象：

```
record = ("risk", [])
record[1].append("ip-1")
```

```
assert record == ("risk", ["ip-1"])
```

元组没有改变所保存的两个对象引用；变化发生在其中的列表对象上。因此“外层不可变”不保证整个对象图都不可变，也不保证一定可哈希。

2. 函数调用是对象共享语义

```
def add_flag(flags: list[str]) -> None:
    flags.append("reviewed")


source = ["created"]
add_flag(source)
assert source == ["created", "reviewed"]
```

调用函数时，实参列表对象绑定到局部名称 `flags`。`append()` 修改共享对象，所以调用者通过 `source` 能看到结果。

如果函数只重新绑定局部名称，调用者不受影响：

```
def replace_flags(flags: list[str]) -> None:
    flags = ["replaced"]


source = ["created"]
replace_flags(source)
assert source == ["created"]
```

面试中只回答“引用传递”容易让人误以为函数能修改调用者变量本身。更准确的表达是：

> Python 以对象共享方式传参。形参是新的局部名称，它绑定到实参求值得到的对象；修改共享可变对象会被外部观察到，重新绑定形参不会改变调用者名称。

3. 可变默认参数

下面的默认列表只在执行 `def`、创建函数对象时求值一次：

```
def collect(value: int, bucket: list[int] = []) -> list[int]:
    bucket.append(value)
    return bucket


assert collect(1) == [1]
assert collect(2) == [1, 2]
```

正确模式是以 `None` 表示“调用者没有提供容器”：

```
def collect(
    value: int,
    bucket: list[int] | None = None,
) -> list[int]:
    if bucket is None:
        bucket = []
    bucket.append(value)
    return bucket
```

默认参数也可能故意用于缓存，但这种隐式状态难以测试和并发控制，通常应使用显式缓存对象或 `functools.cache`。

> 深挖层：`+=` 取决于对象协议
> `x += y` 会优先尝试原地运算协议；对象可以返回自身、返回新对象或返回 `NotImplemented`。列表通常原地扩展，因此共享列表的其他名称也看到变化；元组不可变，通常创建新元组并重新绑定名称。不能仅凭语法符号判断是否修改原对象。

```
left = [1]
alias = left
left += [2]
assert alias == [1, 2]
assert left is alias

numbers = (1,)
old = numbers
numbers += (2,)
assert old == (1,)
assert numbers is not old
```

4. 项目边界

测试平台若把请求模板直接传给用例准备函数并原地补充字段，会污染后续用例：

```
def build_payload(template: dict, case_id: str) -> dict:
    result = dict(template)
    result["case_id"] = case_id
    return result
```

这里的一层复制只隔离顶层字典；如果模板包含嵌套列表或字典，仍要根据修改范围选择深拷贝、不可变数据结构或重新构造。

> 边界案例
> 不要把“不可变对象线程安全”扩大成所有使用场景都安全。多个线程虽然不能修改同一个整数对象，却仍可能竞争地读取、计算并重新绑定共享名称；复合操作是否原子也不能只靠单条源码判断。

5. 面试与练习

问：为什么函数能修改列表，却不能把外部整数改掉？

列表操作修改共享对象；对整数的运算产生新对象并重新绑定局部形参。差异来自对象可变性和具体操作，不是两套参数传递规则。

> 练习
> 1. 解释元组 `("x", [])` 为什么可以观察到内部变化，却仍不能给元组元素重新赋值。
> 2. 分别用列表和元组验证 `+=` 对身份的影响。
> 3. 为可变默认参数缺陷先写失败测试，再完成修复。

官方参考：Function definitions；Data model

A04 is、==、哈希与对象复制

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. `a == b` 一定返回布尔值吗？
2. 两个对象相等时，它们的哈希值应满足什么关系？
3. 为什么包含列表的元组通常不能作为字典键？
4. 浅拷贝隔离了什么，又共享了什么？

> 速记层
> `is` 比较对象身份，不能重载；`==` 调用富比较协议，比较值语义。可哈希对象需要在生命周期内保持哈希稳定，相等对象必须具有相同哈希。浅拷贝新建外层容器但共享内部对象；深拷贝递归复制对象图，并通过 memo 处理共享引用和循环。

1. 身份和值语义

```
a = [1, 2]
b = [1, 2]
c = a

assert a == b
assert a is not b
assert a is c
```

`is` 直接判断是否为同一个对象。`==` 通常触发 `__eq__`，类可以定义“值相等”的业务语义：

```
from dataclasses import dataclass


@dataclass(frozen=True)
class UserKey:
    tenant_id: int
    user_id: int


assert UserKey(1, 2) == UserKey(1, 2)
```

富比较方法可以返回 `NotImplemented`，让解释器尝试反向比较或使用后备行为。`NotImplemented` 是协议返回值，和直接抛出 `NotImplementedError` 不是一回事。

2. 哈希契约

字典和集合先使用哈希缩小候选范围，再用相等判断确认键。自定义对象必须遵守：

> 如果 `a == b` 为真，那么 `hash(a) == hash(b)` 必须为真；反过来不成立，因为不同对象允许发生哈希冲突。

对象作为哈希键后，如果参与相等和哈希计算的字段发生变化，容器可能再也找不到它。因此可变内建容器通常不可哈希：

```
hash([1, 2])  # TypeError
```

元组是否可哈希取决于全部元素：

```
hash((1, "ok"))
hash((1, []))  # TypeError
```

使用 dataclass 时，不要为了“能放进集合”随意打开不安全哈希。更稳妥的起点是不可变值对象：

```
@dataclass(frozen=True)
class RiskKey:
    package_id: str
    region: str
```

3. 浅拷贝

```
import copy

source = {"filters": {"region": ["US"]}}
cloned = copy.copy(source)

assert cloned is not source
assert cloned["filters"] is source["filters"]

cloned["filters"]["region"].append("SG")
assert source["filters"]["region"] == ["US", "SG"]
```

`dict(source)`、`source.copy()`、列表切片等常见写法通常也是浅拷贝。浅拷贝适合只替换顶层字段，不能自动隔离任意嵌套修改。

4. 深拷贝与对象图

```
deep = copy.deepcopy(source)
deep["filters"]["region"].append("JP")

assert source["filters"]["region"] == ["US", "SG"]
```

深拷贝不是简单地“每遇到一次就复制一次”。实现必须维护 memo，才能：

- 保留原对象图中的共享关系。
- 避免循环引用导致无限递归。
- 对不需要复制的对象进行复用。

```
shared = []
graph = [shared, shared]
cloned = copy.deepcopy(graph)

assert cloned[0] is cloned[1]
assert cloned[0] is not shared
```

> 深挖层：相等比较的双向协商
> CPython 的富比较会考虑左、右操作数类型及子类关系，并允许方法返回 `NotImplemented`。这让更具体的子类有机会定义比较语义。阅读入口包括 `Objects/object.c` 的富比较逻辑和具体类型的比较实现。
> 业务类的 `__eq__` 应处理不支持的类型，而不是假设对方总是同类。

5. 项目取舍

接口自动化平台中，完全深拷贝大型请求模板可能消耗大量内存，还可能复制连接、锁或第三方对象失败。更清晰的方案通常是：

- 将模板设计为不可变数据。
- 使用构造函数生成每个用例的数据。
- 只复制将被修改的分支。
- 通过 dataclass 或校验模型显式描述结构。

> 边界案例：**`float("nan")`**
> NaN 不满足普通的自反相等预期：`nan == nan` 通常为假。涉及浮点、时间、数据库 NULL 或业务容差时，需要先定义值语义，不能盲目依赖默认 `==`。

6. 面试与练习

问：为什么重写 **`__eq__`** 后对象可能不能哈希？

因为相等语义改变后，继承来的身份哈希可能违反“相等对象哈希相同”的契约。Python 会在一些情况下把 `__hash__` 设为不可用，要求开发者明确设计不可变性和哈希语义。

> 练习
>
> 1. 实现一个不可变 `CaseKey`，让相同项目和用例 ID 的实例可作为字典键。
> 2. 构造包含循环引用的列表，验证 `deepcopy` 不会无限递归。
> 3. 为嵌套请求模板设计“只复制修改路径”的实现与测试。

官方参考：Object hash contract；copy

A05 数字、布尔值与 None

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. Python 的整数为什么通常不会像固定宽度整数一样溢出？
2. 为什么 `0.1 + 0.2 != 0.3`？
3. 金额计算为什么应优先考虑 Decimal，并从字符串构造？
4. `bool` 与 `int` 有什么关系？

> 速记层
> Python `int` 在内存允许范围内支持任意精度；`float` 通常使用二进制双精度浮点，不能精确表示所有十进制小数；`Decimal` 适合需要明确十进制规则的场景。`bool` 是 `int` 的子类，但业务代码不应利用这一点制造含糊接口。`None` 是表示“没有值”的单例，使用 `is None` 判断。

1. 整数与除法

```
big = 2 ** 10_000
assert isinstance(big, int)
```

```
assert 7 / 2 == 3.5
assert 7 // 2 == 3
assert -7 // 2 == -4
assert -7 % 2 == 1
```

`//` 是向负无穷取整，不是简单截断小数部分。Python 保持关系 `a == (a // b) * b + a % b`，理解负数边界对分页、分桶和时间计算很重要。

> 深挖层：大整数的代价
> CPython 的整数对象以多个“数字位”表示绝对值并记录符号信息，位数会随数值增大。任意精度避免固定宽度溢出，但不代表运算成本恒定：大整数占用更多内存，加法、乘法和字符串转换的成本会随位数增长。源码入口是 `Objects/longobject.c` 和相关头文件。

2. 二进制浮点

```
value = 0.1 + 0.2
print(value)          # 通常显示 0.30000000000000004
assert value != 0.3
```

很多有限十进制小数在二进制中是无限循环，浮点只能保存邻近可表示值。测试计算结果时应根据问题设置容差：

```
import math

assert math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9, abs_tol=0.0)
```

容差不是固定魔法数字。绝对容差适合接近零的比较，相对容差适合随量级变化的误差；金融规则、统计分析和科学计算应采用各自领域的误差模型。

3. Decimal 的构造与上下文

```
from decimal import Decimal, ROUND_HALF_UP

price = Decimal("19.90")
count = 3
total = price * count
display = total.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)

assert display == Decimal("59.70")
```

避免先经过不精确的 float：

```
Decimal(0.1)      # 保留 float 已有的二进制近似
Decimal("0.1")    # 明确的十进制输入
```

Decimal 的精度和舍入受上下文影响。跨服务传输金额时还要统一单位、精度、舍入时点和序列化格式，不能只说“用了 Decimal 就绝对正确”。

4. 布尔值、真假判断和短路

```
assert isinstance(True, int)
assert True + True == 2

name = ""
if not name:
```

```
    ...
```

对象的真假值由 `__bool__` 或 `__len__` 协议决定。`and`、`or` 返回参与运算的对象之一，不一定返回布尔值：

```
configured = "" or "default"
assert configured == "default"
```

用 `x or default` 时要确认 `0`、空容器和空字符串是否也应被视为缺失。如果只有 `None` 表示缺失，应显式判断。

5. `None`、NaN 与数据库 NULL

`None` 是 Python 单例，使用 `is None`。NaN 是浮点值，通常不等于自身：

```
nan = float("nan")
assert nan != nan
assert math.isnan(nan)
```

数据库 NULL、JSON `null` 和 Python `None` 经常互相映射，但三者处于不同语义系统。SQL 中 `NULL = NULL` 不是普通真值，查询应使用 `IS NULL`；业务层还要区分“未提供”“未知”和“明确清空”。

> 项目映射
> 提审服风险指标里的 Top1 占比、HHI、P50/P90/P99 要先确定分母为零、空样本、舍入和展示精度。计算层可保留足够精度，展示层再格式化；不要在每个中间步骤反复四舍五入。

6. 面试与练习

问：Python int 不溢出，是否意味着永远安全？

不是。它仍受内存和时间限制；与数据库固定宽度字段、网络协议、C 扩展交互时仍可能发生范围错误。

> 练习
>
> 1. 为百分比函数定义空分母策略，并分别测试 0、None 和正常值。
> 2. 比较 `round()` 与 Decimal 指定舍入模式在 `.5` 边界上的行为。
> 3. 为浮点统计函数设计合理的 `math.isclose` 测试。

官方参考：Floating-point arithmetic；decimal

A06 字符串、字节与编码

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. Python 3 的 `str` 保存字符还是 UTF-8 字节？
2. `len("中")` 和 UTF-8 编码后的长度为什么不同？
3. 解码失败时，什么时候可以替换字符，什么时候必须失败？
4. 两个视觉相同的字符串为什么可能 `!=`？

> 速记层
> `str` 是 Unicode 文本，`bytes` 是字节序列。编码把文本变成字节，解码把字节解释为文本；边界处必须明确字符集和错误策略。字符、Unicode 码点、用户看到的字形和编码字节不是同一层概念。

1. 文本与字节的边界

```
text = "中国"
payload = text.encode("utf-8")
restored = payload.decode("utf-8")

assert isinstance(text, str)
assert isinstance(payload, bytes)
assert restored == text
assert len(text) == 2
assert len(payload) == 6
```

文件、网络和数据库驱动最终都要处理字节。程序内部尽早解码为 `str`，输出边界再编码，能够减少“某一层到底是什么编码”的混乱。

2. 编解码错误策略

```
raw = b"name:\xff"

raw.decode("utf-8")                    # UnicodeDecodeError
raw.decode("utf-8", errors="replace") # 'name:\uFFFD'
```

`replace` 能让日志或展示继续进行，但会丢失原始信息。需要校验签名、解析协议、入库审计或支持可逆恢复时，静默替换可能破坏证据，应保留原始字节并明确失败。

常见策略包括 `strict`、`replace`、`ignore` 和 `surrogateescape`。`ignore` 会无声删除数据，除非业务明确接受，否则不应成为默认修复方案。

3. Unicode 规范化

同一个视觉字符可能有不同码点序列：

```
import unicodedata

left = "é"
right = "e\u0301"

assert left != right
assert unicodedata.normalize("NFC", left) == unicodedata.normalize("NFC", right)
```

是否规范化取决于业务。用户名搜索可能需要统一，密码和签名输入通常不能擅自改变。大小写无关匹配可考虑 `casefold()`，但同样要结合语言和安全边界。

4. 长度不等于用户看到的字符数

```
family = "\U0001F468\u200d\U0001F469\u200d\U0001F467\u200d\U0001F466"
print(len(family))
```

这个表情由多个码点和连接符组合。`len(str)` 返回实现语言语义中的字符串长度，不等于用户感知的字形数量，也不等于显示宽度。终端对齐、短信计费、数据库长度限制和 UI 截断都需要使用对应领域的规则。

> 深挖层：PEP 393 与灵活字符串表示
> 现代 CPython 会根据字符串中所需的最大码点选择紧凑内部表示，而不是始终为每个字符使用相同的最大宽度。字符串仍表现为不可变 Unicode 序列；内部布局是性能实现，源码入口包括 `Objects/unicodeobject.c` 和相关头文件。不要通过 C 结构猜测跨版本稳定内存格式。

5. 文件、JSON 与数据库

```
from pathlib import Path

path = Path("report.json")
path.write_text('{"status": "正常"}', encoding="utf-8")
content = path.read_text(encoding="utf-8")
```

显式编码让行为不依赖操作系统默认值。JSON 文本有自己的转义规则；HTTP 还需要正确的 `Content-Type` 与 charset；MySQL 连接、数据库和字段字符集也需要一致。出现乱码时按链路逐段检查原始字节和每次编解码，避免通过反复 `encode().decode()` 碰运气。

> 边界案例：路径并不总是纯文本
> Unix 文件名本质上接近不含 NUL 和斜杠的字节序列，Python 使用文件系统编码和错误处理把它呈现为 `str`。处理无法正常解码的路径时，`os.fsencode()`、`os.fsdecode()` 和 surrogate escape 规则比强制 UTF-8 更可靠。

6. 项目与面试

飞书卡片、MySQL 查询结果和 MCP 结构化返回经过多层序列化。日志中出现 `\u4e2d` 可能只是 JSON 转义，不等于乱码；出现 U+FFFD replacement character 则通常意味着某处已不可逆解码失败。排查时应保存原始响应字节、Header 和库版本。

问：Python 3 字符串底层是 UTF-8 吗？

建议回答：语言层的 `str` 是 Unicode 文本，不承诺 UTF-8 内部表示。UTF-8 是常用外部编码；CPython 会采用自己的灵活内部存储。

> 练习
> 1. 比较一个中文字符的 `len(str)`、UTF-8 字节长度和 UTF-16 字节长度。
> 2. 构造组合字符，验证 NFC 规范化前后的相等性。
> 3. 设计一个保留原始字节的解码错误日志结构，避免只记录替换后的文本。

官方参考：Unicode HOWTO；PEP 393

A07 列表、元组、range 与切片

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. 列表尾部追加为什么是“摊销 O(1)”而不是永远 O(1)？
2. `items[:]` 是深拷贝吗？
3. `range(10**12)` 是否会立即占用海量内存？
4. 为什么频繁从列表头部删除通常不是好选择？

> 速记层
> 列表是可变动态数组，适合顺序存储、按下标访问和尾部追加；元组是不可变序列，适合固定记录和稳定接口；`range` 惰性表示等差整数序列；切片通常创建新序列，但内部元素仍然共享。

1. 列表是对象引用数组

列表存放的是对对象的引用，不是把每个对象内容嵌入列表槽位：

```
row = {"id": 1}
rows = [row, row]

rows[0]["status"] = "ok"
assert rows[1]["status"] == "ok"
```

按下标访问只需要定位一个槽位，通常是 O(1)。中间插入或删除需要移动后续引用，通常是 O(n)。尾部 `append()` 在已有容量足够时很快，容量不足时会申请更大的数组并复制槽位，所以只能说摊销 O(1)。

> 深挖层：过度分配
> CPython 列表通常预留一定额外容量，以减少每次追加都重新分配的成本。增长公式是实现细节并会调整；工程判断只需要掌握动态数组的摊销模型。源码入口为 `Objects/listobject.c`，可以关注分配容量与逻辑长度为何分开。

2. 切片规则

```
values = [0, 1, 2, 3, 4]

assert values[1:4] == [1, 2, 3]
assert values[::-1] == [4, 3, 2, 1, 0]
assert values[::2] == [0, 2, 4]
```

切片边界允许超出范围而不抛出 `IndexError`，但单个下标访问会检查范围。列表切片创建新的外层列表，属于浅复制：

```
nested = [[1], [2]]
cloned = nested[:]
cloned[0].append(9)

assert nested[0] == [1, 9]
```

大列表的切片会复制大量引用，时间和额外内存通常为 O(k)，其中 k 是切片长度。只需要迭代窗口时，可以

考虑索引范围、迭代器或专门的数据处理工具。

3. 元组的语义

元组适合表示数量和位置固定的记录：

```
point = (120.1, 30.2)
longitude, latitude = point
```

但字段较多时，位置语义容易出错，应考虑 dataclass、NamedTuple 或校验模型。元组不可变并不自动意味着所有内容可哈希，也不意味着内部对象无法变化。

单元素元组依赖逗号：

```
assert isinstance((1,), tuple)
assert isinstance((1), int)
```

4. `range` 的惰性表示

```
huge = range(0, 10**12, 2)
assert huge[3] == 6
assert 10 in huge
```

`range` 保存起点、终点、步长和长度等有限信息，不会预先创建全部整数。成员判断可以利用算术关系，而不必从头遍历。需要真实列表时再显式转换，但要先评估规模。

5. 队头操作与 `deque`

```
from collections import deque

queue = deque(["a", "b"])
queue.append("c")
assert queue.popleft() == "a"
```

频繁 `list.pop(0)` 会移动后续元素；双端队列适合两端追加和弹出。它不适合替代所有列表，因为随机索引和连续存储行为不同。

> 项目映射
> 自动化平台保存执行队列时，内存中的 `deque` 只适合单进程临时调度。需要跨进程、持久化、重试和可观测性时，应使用数据库或消息队列，不能因为 API 都叫 queue 就忽略可靠性差异。

6. 面试与练习

问：列表和元组只有“能不能修改”的区别吗？

不是。它们表达的设计意图、可哈希可能性、可用方法和某些内存性能特征都不同。固定记录优先考虑具名结构，动态集合使用列表。

> 练习
>
> 1. 验证列表切片是浅复制，并写出不共享嵌套列表的方案。
> 2. 用 `deque` 实现固定长度的最近失败记录。

> 3. 解释 `range(10, 0, -2)` 的终点为什么不包含在结果中。

官方参考：Sequence types；deque

A08 字典、集合与哈希表

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 字典查找为什么只能说平均 O(1)？
2. Python 字典保持插入顺序意味着它是排序映射吗？
3. 字典视图是否是创建时的静态快照？
4. 集合去重后为什么不应依赖任意“业务顺序”？

> 速记层
> 字典把可哈希键映射到值，集合只保留唯一键。哈希先定位候选区域，相等比较确认键；平均查找接近 O(1)，但受哈希质量、冲突、扩容和恶意输入影响。现代 Python 字典保持插入顺序，但不会按键大小自动排序。

1. 查找流程

```
user = {"id": 7, "name": "Lee"}

assert user["id"] == 7
assert user.get("missing") is None
```

概念上的查找步骤是：

1. 计算键的哈希值。
2. 根据哈希定位探测位置。
3. 处理冲突并比较候选键是否相等。
4. 返回对应值或判定不存在。

平均 O(1) 不等于永远一步完成。扩容、冲突、昂贵的 `__hash__` / `__eq__`，以及专门构造的攻击输入都会改变实际成本。

2. 缺失键的语义

`mapping[key]` 在缺失时抛出 `KeyError`；`get()` 返回默认值。两者不是风格差异，而是业务语义：

```
config = {"timeout": None}

assert config.get("timeout") is None
assert config.get("missing") is None
```

上面无法区分“键存在且值为 None”和“键不存在”。需要区分时使用成员判断或专用哨兵：

```
MISSING = object()
value = config.get("missing", MISSING)
assert value is MISSING
```

`setdefault()` 会在缺失时写入，不能把它当成只读查询。聚合数据时 `collections.defaultdict` 可能更清楚，但要注意访问缺失键会改变容器。

3. 顺序与字典视图

字典迭代保持插入顺序：更新已有键通常不改变位置，删除后重新插入会出现在末尾。这个保证不等于按键排序，也不意味着来自数据库或分布式合并的数据天然有确定业务顺序。

```
data = {"a": 1, "b": 2}
keys = data.keys()
data["c"] = 3
assert list(keys) == ["a", "b", "c"]
```

`keys()`、`values()`、`items()` 返回动态视图，不是列表快照。需要冻结当时结果时显式 `list(data.items())`。

4. 集合运算

```
expected = {"create", "update", "delete"}
actual = {"create", "update", "extra"}

missing = expected - actual
unexpected = actual - expected
common = expected & actual
```

集合适合权限差异、字段差异和去重。若输出需要稳定顺序，应在输出边界排序：

```
for name in sorted(missing):
    print(name)
```

5. 字典合并的覆盖方向

```
defaults = {"timeout": 3, "retries": 1}
custom = {"timeout": 10}

settings = defaults | custom
assert settings == {"timeout": 10, "retries": 1}
```

右侧同名键覆盖左侧。配置合并必须明确层级和允许覆盖的字段；把密钥、权限或数据库地址无条件交给外部配置覆盖会形成安全问题。

> 深挖层：紧凑字典与扩容
> CPython 字典实现会将索引信息与条目存储组织起来，以兼顾空间、查找性能和插入顺序。表需要保留空余以控制冲突，达到阈值后扩容。具体负载比例和探测细节是版本实现，源码入口为 `Objects/dictobject.c`；工程上应依赖公开复杂度和顺序语义，而不是内部槽位。

> 边界案例：可变哈希键
> 自定义键如果放入字典后改变了参与哈希或相等判断的字段，键可能仍在容器里却无法按新旧值正常找到。键对象应采用稳定、不可变的身份字段。

6. 项目与面试

风险排查工具可用字典按 `user_id` 聚合多表记录，用集合去重 IP 或地区。但聚合前必须统一键类型：字符串 `"7"` 和整数 `7` 是两个不同键，静默混用会把同一用户拆成两组。

问：字典为什么快？

建议回答：哈希把键映射到有限候选位置，平均避免线性扫描，再用相等比较确认。性能依赖哈希分布、负载和冲突，因此平均 O(1) 不是最坏情况保证。

> 练习
>
> 1. 用集合生成接口字段的 missing、unexpected 和 common 报告。
> 2. 构造存在值为 None 的字典，使用哨兵区分缺失键。
> 3. 实现按用户 ID 聚合登录次数，并测试字符串和整数键混用的失败场景。

官方参考：dict；set

A09 容器选择、推导式、排序与复杂度

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. 推导式一定比普通循环更好吗？
2. Python 排序为什么强调“稳定”？
3. 多条件排序时，如何避免在比较阶段重复计算昂贵字段？
4. 时间复杂度相同的两种方案，实际性能为什么仍可能差很多？

> 速记层
> 先按访问模式选择容器，再考虑语法简短。推导式适合无副作用的简单映射和过滤；复杂分支使用普通循环。`sorted()` 返回新列表，`list.sort()` 原地排序；Python 排序稳定，`key` 函数通常对每个元素计算一次。

1. 从操作模式选容器

需求	常见起点	不适合的信号
保持顺序、按位置访问	`list`	高频队头插删
固定记录	dataclass / tuple	字段多且只靠位置识别
按键查找	`dict`	需要范围排序查询
成员判断、集合差异	`set`	需要重复项或业务顺序
两端队列	`deque`	需要高效随机索引
按优先级取最小项	`heapq`	需要随时获得完整排序结果

复杂度只是第一层。还要考虑对象数量、缓存局部性、序列化成本、可读性和并发边界。

2. 推导式的使用边界

```
active_ids = [
    row["user_id"]
    for row in rows
    if row["status"] == "active"
]
```

当表达式包含多层异常处理、日志、副作用或多个临时变量时，普通循环更容易调试：

```
active_ids = []
for row in rows:
    if row.get("status") != "active":
        continue
    user_id = normalize_user_id(row["user_id"])
    active_ids.append(user_id)
```

生成器表达式适合单次流式消费，列表推导式则立即创建全部结果。不要为了“省内存”返回只能遍历一次的生成器，却让调用方误以为它是可重复读取的集合。

3. 稳定排序与 key

```
rows = [
    {"region": "US", "score": 9},
    {"region": "SG", "score": 9},
    {"region": "JP", "score": 7},
]

ranked = sorted(rows, key=lambda row: row["score"], reverse=True)
```

稳定排序意味着键相等的元素保持原相对顺序。可以利用稳定性执行多阶段排序，也可以直接使用元组键：

```
ranked = sorted(
    rows,
    key=lambda row: (-row["score"], row["region"]),
)
```

`key` 通常对每个元素计算一次，再根据键排序，这比在比较函数中反复执行昂贵解析更可控。

> 深挖层：Timsort 与已有顺序
> CPython 的列表排序使用稳定排序实现，能够利用输入中已有的有序片段，并对复杂边界做专门处理。语言使用者应依赖“稳定”和 API 契约，不依赖内部最小 run、临时数组等参数。源码入口为 `Objects/listobject.c` 及其排序实现文件。

4. TopN 不一定需要全排序

如果只需要大量数据中的少量最大项，可以使用堆：

```
from heapq import nlargest

top_10 = nlargest(10, rows, key=lambda row: row["login_count"])
```

是否更快要靠数据规模和基准测试。若数据来自 MySQL，通常优先让数据库利用索引、`ORDER BY ... LIMIT` 完成候选筛选，避免把全部行传到 Python 后再排序。

5. 复杂度以外的成本

两个 O(n) 方案可能因为以下因素差异巨大：

- 是否进行网络或磁盘 I/O。
- 是否创建大量临时对象。
- Python 层循环还是 C 实现的内建操作。
- 数据是否能放入 CPU 缓存。
- 是否触发序列化、日志或数据库往返。

先用性能分析定位，再优化主导成本；不要仅凭“大 O 更漂亮”重写清晰代码。

> 项目映射
> Top1/Top3/Top10 指标可先在数据库聚合并限制候选，再由 Python 生成统一结果结构。必须记录排序字段、并列规则、空样本和稳定次序，否则同一数据在不同查询计划下可能产生不同展示结果。

6. 面试与练习

问：**`sorted()`** 和 **`list.sort()`** 怎么选？

前者接受任意可迭代对象并返回新列表，适合保留原顺序；后者原地修改列表并返回 `None`，适合明确拥有该列表且希望减少外层复制的场景。

> 练习
>
> 1. 实现稳定的“分数降序、地区升序”排序，并测试并列项。
> 2. 比较全排序取前十与 `heapq.nlargest` 在不同规模下的耗时。
> 3. 把一个包含副作用的复杂推导式重构成可测试循环。

官方参考：Sorting HOWTO；heapq

A10 条件、循环与流程控制

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 20 分钟

先闭卷回答

1. `and`、`or` 是否一定返回 bool？
2. 循环 `else` 在什么时候执行？
3. `match` 是普通 `switch` 的同义词吗？
4. 为什么宽泛地用 `except` 控制正常流程会掩盖问题？

> 速记层
> 条件判断调用真假值协议；`and`、`or` 短路并返回操作数。循环 `else` 在循环未被 `break` 中断时执行。结构化模式匹配按模式解构对象，不只是值分支。流程控制应让成功路径和失败路径都清晰可测。

1. 真假值与短路

```
def load_config(explicit, fallback):
    return explicit if explicit is not None else fallback
```

如果写成 `explicit or fallback`，空字典、空字符串、0 和 False 都会被当成缺失。选择简写前要先确定业务语义。

短路可以避免不必要计算：

```
if user is not None and user.is_active:
    grant_access(user)
```

但不要在短路表达式里堆叠副作用，例如 `condition and send_message()`，因为返回值含义和执行路径不直观。

2. 循环 `else`

```
def find_user(rows, user_id):
    for row in rows:
        if row["user_id"] == user_id:
            return row
    else:
        return None
```

`else` 在循环正常耗尽、没有执行 `break` 时运行。上例直接 `return` 已很清晰，`else` 并非必需；它在需要区分“找到并中断”和“完整检查后未找到”时有用：

```
for row in rows:
    if invalid(row):
        break
else:
    publish(rows)
```

3. `for` 的迭代协议

`for` 不要求对象是列表。它先取得迭代器，再不断调用下一项，直到收到 `StopIteration`。因此修改正在迭代的列表或字典可能跳过元素或触发错误。需要删除字典键时，可以迭代快照：

```
for key in list(config):
    if key.startswith("deprecated_"):
        del config[key]
```

4. 结构化模式匹配

```
def handle(event: dict) -> str:
    match event:
        case {"type": "login", "user_id": int(user_id)}:
            return f"login:{user_id}"
        case {"type": "logout"}:
            return "logout"
        case _:
            return "unknown"
```

模式匹配可以检查结构、字面值和类型，并绑定局部名称。它不是任意布尔条件的替代品。复杂守卫、隐藏捕获变量和过宽映射模式都可能让代码难读。

> 深挖层：名称模式的陷阱
> 在 `case NAME:` 中，普通裸名称通常是捕获模式，而不是拿已有变量做值比较；常量值模式通常需要限定名称或字面量。第一次使用 match 时应通过最小测试确认每个分支，避免把“想比较”写成“无条件捕获”。

5. 海象运算符的边界

```
while chunk := stream.read(8192):
    process(chunk)
```

赋值表达式适合避免重复计算并让循环条件紧邻结果。若一行中出现多个赋值、条件和函数调用，拆开通常更清楚。

6. 项目与面试

自动化测试调度器的状态流转不应靠深层 `if/elif` 随意修改。状态数量增多时，应把允许的转换集中定义并为非法路径写测试；模式匹配可以改善事件解构，但不能替代状态机约束。

问：循环 **`else`** 在循环一次都没执行时会运行吗？

会，只要循环正常结束且没有被 `break` 中断。空迭代器也是正常耗尽。

> 练习
>
> 1. 为“遍历全部规则且没有拒绝项才通过”分别写循环 `else` 和 `all()` 版本。
> 2. 构造 match 捕获模式误写成常量比较的例子，并用测试暴露问题。
> 3. 找出一段在迭代列表时删除元素的代码，修复并说明复杂度变化。

官方参考：Compound statements；Structural Pattern Matching tutorial

A11 函数调用与参数规则

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. `/` 和 `*` 在函数签名中分别约束什么？
2. `*args` 与 `**kwargs` 在函数内部是什么类型？
3. 默认参数什么时候求值？
4. 为什么公共 API 不应无条件接受并忽略任意 `**kwargs`？

> 速记层
> 调用函数时先计算实参，再按签名把对象绑定到形参。`/` 左侧是仅限位置参数，裸 `*` 右侧是仅限关键字参数；`*args` 收集多余位置参数为元组，`**kwargs` 收集多余关键字参数为字典。签名是接口契约，不只是语法。

1. 完整参数形态

```
def query_risk(
    package_id: str,
    /,
    days: int = 7,
    *,
    include_internal: bool = False,
) -> dict:
    ...
```

- `package_id` 只能按位置传入。
- `days` 可以按位置或关键字传入。
- `include_internal` 只能按关键字传入。

仅限关键字适合布尔开关和单位参数，可避免 `query_risk("pkg", 30, True)` 这种难读调用。仅限位置可以避免把内部形参名永久暴露成 API，并允许未来调整名称。

2. 收集与展开

```
def record(event: str, *tags: str, **fields: object) -> dict:
    return {"event": event, "tags": tags, "fields": fields}


payload = {"user_id": 7, "region": "US"}
result = record("login", "risk", "review", **payload)
```

调用时展开映射要求键适合作为关键字名称，并且不能与显式参数重复。接口边界直接接收任意 `**kwargs` 容易吞掉拼写错误：

```
send(timeout=3, tiemout=5)  # 如果任意 kwargs 被忽略，错误可能静默存在
```

公共接口应验证未知字段，或使用明确的数据模型。

3. 参数绑定发生在函数体之前

缺少参数、重复传值或出现未知关键字时，函数体还没有执行就会抛出 `TypeError`。可以用 `inspect.signature()` 在框架层复用绑定规则：

```
from inspect import signature


def task(case_id: int, *, retry: int = 0) -> None:
    ...


bound = signature(task).bind(7, retry=2)
bound.apply_defaults()
assert bound.arguments == {"case_id": 7, "retry": 2}
```

MCP、依赖注入和命令行框架经常根据签名生成 Schema 或完成绑定，因此装饰器保留签名非常重要。

> 深挖层：Vectorcall
> 现代 CPython 为常见调用路径提供 vectorcall 协议，以减少临时元组和字典分配。它属于调用性能实现，不改变语言的参数绑定语义。源码入口包括对象调用抽象层、函数对象实现和具体可调用类型；不同版本的内部 API 会变化。

4. 默认值与哨兵

默认对象在函数定义时创建。若 `None` 本身也是合法输入，使用独立哨兵：

```
MISSING = object()


def update(value=MISSING):
    if value is MISSING:
```

```
        return "not provided"
    if value is None:
        return "explicitly cleared"
    return "updated"
```

这在 PATCH API、配置合并和 MCP 工具参数中尤其重要，因为“未提供”和“传 null”可能代表不同动作。

5. 项目与面试

风险查询工具的参数应把时间范围、是否包含内网等开关设计为关键字参数，并在入口统一做类型、范围和组合校验。函数签名越清晰，自动生成 MCP Schema 和测试用例越可靠。

问：`*args`、`**kwargs` 会影响性能吗？

可能引入收集和展开成本，但接口清晰度通常更重要。热点路径应先分析调用规模；不能为了微小开销牺牲正确签名，也不能用它们掩盖无边界参数。

> 练习
>
> 1. 把一个含三个布尔位置参数的函数改为仅限关键字参数，并补充错误调用测试。
> 2. 使用哨兵区分“缺失”和“明确传 None”。
> 3. 用 `inspect.signature().bind()` 验证动态工具调用参数。

官方参考：Function definitions；inspect.Signature

A12 作用域、闭包与函数对象

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. LEGB 中的 E 是什么？
2. 为什么在函数内给名称赋值可能导致 `UnboundLocalError`？
3. 闭包捕获的是定义时的值还是名称所在的单元？
4. 循环创建 lambda 时，为什么经常得到相同结果？

> 速记层
> 名称解析通常按 Local、Enclosing、Global、Builtins 查找。编译器会根据代码块中的绑定操作决定局部名称；读取尚未赋值的局部名称会触发 `UnboundLocalError`。闭包让内部函数保留对外层自由变量的访问，通常是延迟查值而不是自动复制当时值。

1. 局部名称由编译阶段决定

```
count = 10


def broken():
    print(count)
    count = 11
```

函数体存在对 `count` 的赋值，编译器会把它视为局部名称；执行 `print(count)` 时局部绑定尚未建立，因此抛

出 `UnboundLocalError`。这不是运行到赋值行后才决定作用域。

若确实要修改模块全局名称，可以声明 `global`，但共享全局状态会增加测试和并发难度。多数场景更适合把状态封装进对象或显式传参。

2. 闭包

```
def make_threshold_checker(threshold: float):
    def check(value: float) -> bool:
        return value >= threshold

    return check


is_high = make_threshold_checker(0.8)
assert is_high(0.9)
```

外层函数结束后，内部函数仍能访问 `threshold`。可以检查闭包单元：

```
assert is_high.__closure__[0].cell_contents == 0.8
```

闭包适合生成小型策略函数、装饰器和回调，但复杂可变状态更适合显式类，因为生命周期和并发控制更清楚。

3. 延迟绑定陷阱

```
checks = [lambda: i for i in range(3)]
assert [fn() for fn in checks] == [2, 2, 2]
```

调用 lambda 时循环已经结束，所有闭包读取同一个 `i`。可以通过默认参数在创建函数时固定对象：

```
checks = [lambda i=i: i for i in range(3)]
assert [fn() for fn in checks] == [0, 1, 2]
```

也可以使用 `functools.partial()`，让意图更明确。

4. `nonlocal`

```
def counter():
    value = 0

    def increment():
        nonlocal value
        value += 1
        return value

    return increment
```

`nonlocal` 修改最近的外层函数作用域绑定，不会查找模块全局。它适合小型封装；多个操作共享复杂状态时，类更容易加入锁、类型和测试接口。

> 深挖层：符号表、cell 与 free variable
> 编译阶段会分析每个代码块中的绑定和引用。被内部函数引用的外层局部名称需要存入 cell，内部代码对象把它视为 free variable。可以观察 `code.co_cellvars`、`code.co_freevars` 和 `__closure__`。这些属性适合学习和调试，但不要让业务逻辑依赖内部单元顺序。

5. 函数是一等对象

函数可以赋值、存入容器、作为参数传递或作为返回值：

```
OPERATIONS = {
    "sum": sum,
    "max": max,
}

result = OPERATIONS["max"]([1, 9, 3])
assert result == 9
```

策略映射比长 `if/elif` 清晰，但外部输入不能直接选择任意模块函数；应使用白名单并验证参数。

> 项目映射
> 自动化平台若在循环中为每个项目创建回调，必须测试回调是否绑定各自项目 ID。延迟绑定缺陷常在任务真正异步执行时才暴露，因为循环早已结束。

6. 面试与练习

问：闭包有什么实际用途？

可以保存配置并生成函数，例如装饰器、校验器和回调；代价是状态隐式、调试和序列化更困难，复杂状态应考虑对象。

> 练习
>
> 1. 复现 `UnboundLocalError`，分别用显式参数、对象状态和 `global` 修复，比较取舍。
> 2. 修复循环 lambda 延迟绑定，并写测试防止回归。
> 3. 检查一个闭包的 `co_freevars` 和 cell 内容。

官方参考：Resolution of names；Function objects

A13 装饰器

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. `@decorator` 在什么时候执行？
2. 为什么装饰器经常导致函数名、文档和签名丢失？
3. 带参数装饰器为什么通常需要三层函数？
4. 同一函数上的多个装饰器按什么顺序应用？

> 速记层
> 装饰器在定义阶段接收被装饰对象，并把名称重新绑定到返回对象。它常用于横切关注点，如计时、权限、重试和注册。包装函数应使用 `functools.wraps` 保留元数据；副作用、同步异步边界和装饰顺序都必须测试。

1. 本质是重新绑定

```
def audit(func):
    def wrapper(*args, **kwargs):
        print("before")
        return func(*args, **kwargs)

    return wrapper


@audit
def query(package_id: str) -> dict:
    return {"package_id": package_id}
```

大致等价于：

```
def query(package_id: str) -> dict:
    return {"package_id": package_id}


query = audit(query)
```

装饰器表达式在定义阶段求值。模块导入时执行的注册装饰器因此可能产生导入副作用。

2. 保留元数据

```
from functools import wraps


def audit(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)

    return wrapper
```

`wraps` 复制常用元数据并设置 `__wrapped__`，方便 `inspect`、测试框架和文档工具找到原函数。它不能自动让任意包装器拥有完全相同的静态类型，复杂装饰器可以结合 ParamSpec 和 TypeVar 描述签名。

3. 带参数装饰器

```
from collections.abc import Callable
from functools import wraps


def retry(times: int):
    if times < 1:
        raise ValueError("times must be positive")

    def decorate(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            last_error = None
            for _ in range(times):
                try:
                    return func(*args, **kwargs)
                except TimeoutError as exc:
                    last_error = exc
            raise last_error

        return wrapper

    return decorate
```

真实重试还需要幂等性、退避、抖动、超时预算、可重试异常白名单和日志；装饰器只解决复用结构，不自动保证策略正确。

4. 顺序

```
@outer
@inner
def run():
    ...
```

应用时近似 `run = outer(inner(run))`，调用时通常先进入 outer 包装器。权限、事务、重试和指标的顺序会改变行为，例如重试包在事务外还是事务内，可能决定每次尝试是否拥有独立事务。

> 深挖层：描述符与方法装饰
> 普通函数作为类属性时会通过描述符协议产生绑定方法。装饰器若返回不实现相同协议的自定义对象，可能改变方法绑定行为；`classmethod`、`staticmethod`、`property` 也都是描述符式包装。装饰顺序错误会导致拿到不同对象类型。

5. 同步与异步边界

同步包装器直接调用异步函数只会得到协程对象，无法捕获实际执行时异常：

```
import inspect


def traced(func):
    if inspect.iscoroutinefunction(func):
        @wraps(func)
        async def async_wrapper(*args, **kwargs):
            return await func(*args, **kwargs)
        return async_wrapper

    @wraps(func)
    def sync_wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return sync_wrapper
```

生产实现还应保留类型、异常和上下文传播。

> 项目映射
> MCP 工具注册本质上常依赖装饰器或显式注册表。注册装饰器除了返回函数，还可能读取注解和 docstring 生成 Schema。若自定义装饰器没有保留 `__wrapped__` 和签名，工具参数可能退化成 `args/kwargs`。

6. 面试与练习

问：装饰器和中间件有什么区别？

装饰器包装具体可调用对象；中间件通常处于框架请求或消息管线。二者都处理横切逻辑，但生命周期、作用范围和上下文不同。

> 练习

1. 写一个同时支持同步和异步函数的计时装饰器。
2. 测试被装饰函数的 `__name__`、签名和 `__wrapped__`。
3. 用两个记录顺序的装饰器验证应用顺序和调用顺序。

官方参考：Decorator glossary；functools.wraps

A14 可迭代对象、迭代器与生成器

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 可迭代对象和迭代器有什么区别？
2. 为什么生成器通常只能消费一次？
3. `yield` 暂停时保存了哪些运行状态？
4. 生成器何时释放文件或数据库游标？

速记层
可迭代对象能产生迭代器；迭代器通过 `__next__` 逐项返回值并以 `StopIteration` 表示结束。生成器是由生成器函数或表达式创建的迭代器，保存暂停点和执行帧状态。惰性减少峰值内存，但会引入一次性消费、资源生命周期和异常时机问题。

1. 两层协议

```
values = [1, 2, 3]
iterator = iter(values)

assert iter(values) is not iter(values)
assert iter(iterator) is iterator
assert next(iterator) == 1
```

列表是可迭代对象，每次 `iter(list)` 可以创建独立迭代器；列表迭代器本身也是迭代器，通常返回自己并记录当前位置。

自定义倒计时迭代器：

```
class Countdown:
    def __init__(self, start: int):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value
```

如果对象需要支持多次独立遍历，应让 `__iter__` 创建新的迭代器，而不是把遍历状态放在容器本身。

2. 生成器函数

```
def read_ids(rows):
    for row in rows:
        if row.get("user_id") is not None:
            yield int(row["user_id"])


ids = read_ids([{"user_id": "7"}, {}])
assert list(ids) == [7]
assert list(ids) == []
```

调用包含 `yield` 的函数会创建生成器对象，不会立即执行函数体。第一次 `next()` 才开始运行；每次 `yield` 返回一个值并暂停，下一次继续。

3. 异常和返回值

生成器正常结束通过 `StopIteration` 表示。生成器函数中的 `return value` 会把 value 放入结束异常中，普通 `for` 会自动处理它。不要在普通生成器主体中手动抛 `StopIteration` 模拟结束；现代 Python 会把不当传播转换为运行时错误。

`yield from` 可以委托子迭代器，并转发发送、异常和返回值协议：

```
def chain(*iterables):
    for iterable in iterables:
        yield from iterable
```

4. 惰性的收益和代价

惰性管道只有在消费时才发生 I/O 和错误：

```
lines = (line.strip() for line in open("events.log", encoding="utf-8"))
```

这个写法隐藏了文件关闭责任。如果生成器没有完全消费，文件可能保持打开。更清楚的方式是让拥有资源的一层同时拥有迭代生命周期：

```
def iter_lines(path):
    with open(path, encoding="utf-8") as stream:
        for line in stream:
            yield line.rstrip("\n")
```

调用者仍应完整消费或显式关闭生成器。对于关键资源，可以使用上下文管理器返回迭代器，或把批处理封装成在函数内部完成。

> 深挖层：暂停的执行帧
> 生成器对象关联代码对象和可恢复帧状态，包括指令位置、局部变量和异常状态。
> `generator.gi_frame`、`gi_code`、`gi_running` 可用于调试；运行完成后部分引用会被释放。内部字段和帧实现随版本变化，源码入口包括生成器对象和解释器执行相关文件。

5. 生成器的控制接口

生成器支持 `send()`、`throw()` 和 `close()`，但双向协程式生成器复杂度很高。现代异步任务优先使用 `async` / `await`；普通数据流生成器保持单向产出更容易维护。

> 边界案例：迭代时修改容器
> 列表迭代器通常按索引前进，修改长度可能跳过或重复业务元素；字典和集合在迭代期间改变大小通常会报错。不要把当前 CPython 的偶然结果当成允许修改的契约。

6. 项目与面试

数据库查询若一次返回百万行，生成器或服务端游标可以降低峰值内存，但事务会持续更久，连接被占用，失败重试也更复杂。应结合批大小、连接池、超时和幂等处理，而不是只强调“生成器省内存”。

问：生成器和列表哪个更快？

没有统一答案。生成器降低内存并推迟计算，列表适合重复访问、长度查询和多次遍历。总耗时还取决于 Python 迭代开销和下游访问模式。

> 练习
>
> 1. 实现可重复迭代的容器和一次性迭代器，比较 `iter()` 行为。
> 2. 写一个分批读取数据的生成器，并测试中途关闭时资源是否释放。
> 3. 使用 `yield from` 合并多个结果源，验证空输入和异常传播。

官方参考：Iterator types；Yield expressions

A15 类、实例、属性与方法

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 实例访问 `obj.method` 时为什么会自动获得 `self`？
2. 类属性被可变对象污染的典型场景是什么？
3. `staticmethod` 和模块函数如何取舍？
4. `property` 是否意味着计算可以任意昂贵？

> 速记层
> 类本身也是对象，调用类通常创建实例。实例属性通常存放每个对象的状态，类属性由类和未覆盖它的实例共享。普通函数放在类属性上时通过描述符协议形成绑定方法，把实例作为第一个参数。方法类型应表达归属和所需状态。

1. 实例与类属性

```
class Job:
    category = "test"

    def __init__(self, job_id: str):
        self.job_id = job_id
```

`job.job_id` 通常来自实例；`job.category` 在实例没有同名属性时从类上找到。给实例重新赋值会遮蔽类属性：

```
first = Job("a")
second = Job("b")
first.category = "risk"

assert first.category == "risk"
assert second.category == "test"
assert Job.category == "test"
```

可变类属性会被所有未覆盖实例共享：

```
class BrokenJob:
    events = []
```

除非共享是明确设计，否则应在 `__init__` 中创建实例容器。

2. 方法绑定

```
class Counter:
    def increment(self, value: int) -> int:
        return value + 1


counter = Counter()
bound = counter.increment

assert bound(1) == 2
assert bound.__self__ is counter
```

从实例访问普通函数时会得到绑定方法，内部保存实例和原函数。通过类调用则需显式传实例：

```
assert Counter.increment(counter, 1) == 2
```

> 深挖层：函数描述符
> 函数对象实现描述符协议。属性查找发现类中的函数后，会根据访问者产生绑定方法；CPython 的方法对象在调用时把 `self` 放到参数前面，并使用高效调用协议。源码入口包括 `Objects/funcobject.c`、`Objects/classobject.c` 和属性访问实现。

3. 三种方法

```
class RiskScore:
    scale = 100

    def normalized(self, raw: float) -> float:
        return raw / self.scale

    @classmethod
    def from_ratio(cls, ratio: float):
        return cls(ratio * cls.scale)

    @staticmethod
    def validate(raw: float) -> None:
        if raw < 0:
            raise ValueError("raw must be non-negative")
```

- 实例方法需要实例状态或行为。
- 类方法常用于替代构造器和可继承的类级策略。
- 静态方法不接收实例或类，只是放在类命名空间中。

若函数不依赖类概念，模块函数通常更直接。不要用 `staticmethod` 只为“看起来面向对象”。

4. `property`

```
class QueryWindow:
    def __init__(self, days: int):
        self._days = days

    @property
    def days(self) -> int:
        return self._days

    @days.setter
    def days(self, value: int) -> None:
        if value < 1:
            raise ValueError("days must be positive")
        self._days = value
```

property 可以在保留属性式接口时加入校验或计算。调用方通常预期属性访问便宜且无明显副作用；网络请求、数据库写入等动作应使用方法名明确表达。

5. `__slots__` 的边界

`__slots__` 可以限制常见实例属性存储并在大量小对象时减少部分内存，但会影响继承、弱引用、动态属性和某些工具兼容性。先通过内存分析确认对象数量和瓶颈，再采用。

> 项目映射
> 测试平台可以把执行上下文建模为实例，把跨全部任务共享且不可变的常量留在类或模块层。浏览器、数据库连接和可变结果不能放进无隔离的类属性，否则并发 Worker 会互相污染。

6. 面试与练习

问：类方法和静态方法有什么区别？

类方法接收实际调用类 `cls`，适合可继承构造和类级行为；静态方法不接收隐式对象，只表达命名归属。

> 练习
>
> 1. 复现可变类属性被两个实例共享的问题并修复。
> 2. 观察绑定方法的 `__self__` 和 `__func__`。
> 3. 把一个执行数据库查询的 property 重构为意图明确的方法。

官方参考：Classes tutorial；Descriptor HOWTO

A16 继承、组合、MRO 与 `super`

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. `super()` 是否等于“调用父类”？
2. 多继承中的方法查找顺序如何确定？
3. 为什么 mixin 的协作方法通常要继续调用 `super()`？
4. 什么时候组合比继承更稳妥？

> 速记层
> 继承表达 is-a 和可替换关系，组合表达对象拥有或使用另一个能力。Python 根据 C3 线性化得到 MRO；`super()` 沿当前 MRO 的下一个实现继续协作，不是写死某个“父类”。多继承要求方法签名和初始化链可协作。

1. MRO

```
class Root:
    def run(self):
        return ["root"]


class LoggingMixin(Root):
    def run(self):
        return ["log", *super().run()]


class TimingMixin(Root):
    def run(self):
        return ["time", *super().run()]


class Worker(LoggingMixin, TimingMixin):
    pass


assert Worker().run() == ["log", "time", "root"]
print(Worker.__mro__)
```

`super()` 在 `LoggingMixin` 中并不固定跳到源码中写出的某个父类，而是根据实际实例类型和 MRO 找下一个实现。这使 mixin 可以组合，也要求每一层都合作。

2. 初始化链

协作式多继承常让每层只消费自己的参数，再把其余参数传下去：

```
class Named:
    def __init__(self, *, name: str, **kwargs):
        self.name = name
        super().__init__(**kwargs)
```

这种模式对签名一致性要求很高。若某一层不调用 `super()`，链条会中断；若参数命名冲突，初始化可能失败。简单业务对象不要为复用几行代码引入复杂菱形继承。

3. 可替换性

子类应能放在基类预期位置而不破坏契约。常见违反包括：

- 收紧输入条件。
- 改变返回值语义。
- 抛出调用方不预期的新异常。
- 把原本只读操作变成有副作用操作。

如果子类只是“碰巧复用代码”，却不是同一抽象，组合通常更清晰：

```
class RiskService:
    def __init__(self, repository, notifier):
        self.repository = repository
        self.notifier = notifier
```

服务使用仓库和通知器，不需要继承它们。

> 深挖层：C3 线性化
> MRO 需要保留每个父类的局部先后关系，并产生一致的单调顺序。出现不可线性化的继承图时，类创建阶段会失败，而不是运行时随意选路。可用 `Class.__mro__` 或 `inspect.getmro()` 检查，不要凭源码缩进猜测。

4. Mixin 的约束

好的 mixin 通常：

- 职责单一，不独立实例化。
- 名称明确以 Mixin 结尾。
- 少持有状态，或清楚声明依赖。
- 方法签名能与协作链兼容。
- 有组合顺序测试。

> 边界案例
> 直接调用 `Base.method(self)` 会绕开 MRO 后续节点，可能让其他 mixin 永远得不到执行。只有明确需要跳过协作链且能证明安全时才这样做。

5. 项目与面试

自动化框架常把截图、重试、日志设计成基类层层继承，最终难以判断哪一层拥有浏览器和清理责任。更容易维护的结构是通过 fixture、组合对象或中间件显式拼装能力。

问：**`super()`** 是调用父类吗？

建议回答：它返回一个代理，从当前类在实际对象 MRO 中的位置之后继续查找；单继承时常看起来像父类调用，多继承时“下一个类”不一定是源码中直接父类。

> 练习

1. 建立两个 mixin 和根类，打印 MRO 并验证调用顺序。
2. 故意让某层不调用 `super()`，写测试观察链条中断。
3. 把一个多层 Service 继承结构重构为组合。

官方参考：Multiple inheritance；super

A17 对象协议、dataclass、ABC、Protocol、描述符与元类

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 30 分钟

先闭卷回答

1. 鸭子类型依赖继承关系吗？
2. ABC 与 typing.Protocol 分别解决什么问题？
3. 数据描述符为什么可能优先于实例字典？
4. 元类在什么时候介入？普通业务为什么通常不需要自定义元类？

> 速记层
> Python 通过特殊方法协议让对象参与长度、迭代、上下文、比较和调用等语法。dataclass 生成常见数据方法；ABC 提供名义接口和运行时抽象约束；Protocol 支持静态结构化子类型。描述符控制属性访问，函数、property、classmethod 都建立在该协议上；元类控制类对象的创建。

1. 特殊方法是协议入口

```
class Batch:
    def __init__(self, rows):
        self._rows = list(rows)

    def __len__(self):
        return len(self._rows)

    def __iter__(self):
        return iter(self._rows)


batch = Batch([1, 2])
assert len(batch) == 2
assert list(batch) == [1, 2]
```

应通过 `len(batch)`、`iter(batch)` 使用协议，而不是手动调用特殊方法。解释器对特殊方法查找有专门规则，实例上临时放同名属性不一定影响运算符行为。

2. dataclass 是代码生成，不是校验框架

```
from dataclasses import dataclass, field


@dataclass(frozen=True, slots=True)
class QuerySpec:
    package_id: str
    regions: tuple[str, ...] = field(default_factory=tuple)
```

dataclass 可生成初始化、表示、比较等方法。`default_factory` 避免共享可变默认值。类型注解默认不做运行时校验；来自 HTTP 或 MCP 的不可信输入仍需要验证。

`frozen=True` 阻止普通字段赋值，但不是密码学不可变，也无法保证成员对象不可变。

3. ABC 与 Protocol

```
from typing import Protocol
```

```
class RiskRepository(Protocol):
    def fetch(self, package_id: str) -> list[dict]: ...


def build_report(repository: RiskRepository, package_id: str):
    return repository.fetch(package_id)
```

任意具有兼容 `fetch` 方法的对象都可被静态检查器接受，不需要显式继承 Protocol。ABC 则适合需要共享实现、运行时 `isinstance` 语义或明确注册体系的场景。

结构化类型降低耦合，但接口不能只看方法名，还要约定异常、性能、事务和生命周期。

4. 描述符与属性优先级

```
class Positive:
    def __set_name__(self, owner, name):
        self.private_name = f"_{name}"

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return getattr(instance, self.private_name)

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("must be positive")
        setattr(instance, self.private_name, value)
```

实现 `__get__`、`__set__` 或 `__delete__` 的对象可以控制另一个类的属性访问。数据描述符通常优先于实例字典，非数据描述符可能被实例属性遮蔽。ORM 字段、property 和函数方法绑定都利用了描述符思想。

> 深挖层：属性查找的大致顺序
> 对常见实例访问，可概括为：先考虑类及 MRO 中的数据描述符，再看实例存储，再看非数据描述符或普通类属性，最后可能进入 `__getattr__`。自定义 `__getattribute__` 会接管全部访问，稍有错误就会递归。源码入口是 `Objects/object.c` 和 `Objects/typeobject.c`。

5. 元类

类对象由元类创建，普通类默认使用 `type`。元类可以在类创建阶段修改命名空间、注册类或验证声明，但会增加阅读和工具兼容成本。许多需求可由类装饰器、`__init_subclass__`、描述符或显式注册表解决。

```
class Plugin:
    registry = {}

    def __init_subclass__(cls, *, name: str, **kwargs):
        super().__init_subclass__(**kwargs)
        Plugin.registry[name] = cls
```

这通常比自定义元类更容易理解。

> 项目映射
> 测试平台可用 Protocol 定义执行器和结果仓库，使单元测试传入内存替身；MCP 工具注册可用装饰器或 `__init_subclass__`。只有框架级类声明系统确实需要控制类创建时，再评估元类。

6. 面试与练习

问：Protocol 和 ABC 怎么选？

如果希望已有对象只要结构兼容就能使用，并以静态检查为主，Protocol 更灵活；如果需要共享实现、实例化限制或明确运行时类型关系，ABC 更适合。二者也可以组合。

> 练习
>
> 1. 为数据库仓库定义 Protocol，并实现内存替身测试服务。
> 2. 实现一个正整数描述符，测试类访问、实例访问和非法赋值。
> 3. 把一个元类注册示例改为 `__init_subclass__`，比较复杂度。

官方参考：Special method names；dataclasses；Protocol

A18 异常、上下文管理与文件处理

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. `else` 和 `finally` 在异常结构中分别什么时候执行？
2. 为什么 `except Exception: pass` 危险？
3. `raise ... from ...` 解决什么问题？
4. 上下文管理器是否等于“只处理文件”？

> 速记层
>
> 异常把失败沿调用栈传播，捕获层应有处理能力或补充语义。只捕获预期异常，并保留原因链；`finally` 用于无论成功失败都要执行的清理。上下文管理器把获取与释放组成协议，适用于文件、锁、事务、临时配置和追踪区间。

1. 完整结构

```
try:
    payload = read_payload()
except UnicodeDecodeError as exc:
    raise InvalidPayload("payload is not UTF-8") from exc
else:
    process(payload)
finally:
    metrics.increment("attempts")
```

- `except` 处理匹配异常。
- `else` 只在 try 正常完成时执行，可缩小捕获范围。
- `finally` 无论是否异常都执行，除非进程被强制终止等特殊情况。

不要把可能抛出其他异常的大段业务代码全部放进 try，再误当作读取错误捕获。

2. 异常转换与原因链

```
class RepositoryUnavailable(RuntimeError):
    pass


def fetch(connection):
    try:
        return connection.query()
    except TimeoutError as exc:
        raise RepositoryUnavailable("risk repository timed out") from exc
```

上层获得领域语义，同时 Traceback 保留底层原因。若故意隐藏实现异常，可以 `raise PublicError(...) from None`，但应谨慎，日志或内部观测仍需保留证据。

3. 捕获边界

`Exception` 不包含所有 `BaseException` 子类，通常不会吞掉 `KeyboardInterrupt`、`SystemExit`。即便如此，宽泛捕获也只适合任务边界、请求边界等必须记录失败并隔离的地方，而且应重新抛出或转换：

```
try:
    run_one_job(job)
except Exception:
    logger.exception("job failed", extra={"job_id": job.id})
    mark_failed(job)
```

不能记录后继续假装成功。

4. 上下文管理器

```
from contextlib import contextmanager


@contextmanager
def transaction(connection):
    try:
        yield connection
    except Exception:
        connection.rollback()
        raise
    else:
        connection.commit()
```

`with` 调用进入协议获得资源，在退出时把异常信息交给退出协议。退出方法返回真值可以抑制异常，因此自定义管理器必须明确是否应该吞异常。

> 深挖层：异常会保留对象图
> Traceback 引用栈帧，栈帧引用局部变量，长时间保存异常对象可能间接保留大量数据。任务队列若把完整异常对象放入全局列表，会造成内存增长；通常应保存结构化摘要、格式化 Traceback 和必要上下文。

5. 文件与原子写入思路

```
from pathlib import Path


def load_text(path: Path) -> str:
    return path.read_text(encoding="utf-8")
```

写关键配置时，直接覆盖目标文件可能在进程中断后留下半文件。常见策略是在同一文件系统写临时文件、刷新并原子替换；还要处理权限、备份和并发写入。`tempfile` 和 `os.replace()` 是实现入口，但“原子”仍受文件系统和跨设备边界影响。

6. 异常组与并发

现代 Python 的并发结构可能同时产生多个异常，以 ExceptionGroup 表示，并可通过 `except*` 按类型处理。不要把它硬压成“第一个错误”，否则会丢失并发任务的其他失败证据。Python 3.11 是异常组和 TaskGroup 的重要版本基线。

> 项目映射
> 自动化平台需要区分用例断言失败、测试数据错误、浏览器超时、平台基础设施故障。统一显示“执行失败”会让重试、统计和责任定位都失真。异常类型应服务于可恢复策略和用户可理解结果。

7. 面试与练习

问：什么时候自定义异常？

当调用方需要按领域语义处理、隐藏底层实现或形成稳定错误契约时。异常层级应简洁，不要为每条错误消息创建新类型。

> 练习
> 1. 实现事务上下文管理器，测试成功提交、失败回滚和原异常保留。
> 2. 把宽泛 try 块缩小到真正预期失败的操作。
> 3. 构造 TaskGroup 多任务失败，观察 ExceptionGroup 的结构。

官方参考：Errors and Exceptions；contextlib；Exception groups

A19 模块、包、导入与项目结构

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. 导入模块时，模块顶层代码是否执行？
2. `sys.path` 与已导入模块缓存分别解决什么问题？
3. 循环导入为什么有时失败、有时又看似能运行？
4. `pyproject.toml` 能承担哪些项目职责？

> 速记层
> 模块是命名空间和代码组织单元，包组织模块。导入先解析名称并寻找规范，再加载、创建模块对象、执行代码，结果通常缓存在 `sys.modules`。项目应使用明确包边界、单一配置入口、隔离环境和锁定依赖，避免依赖当前工作目录和导入副作用。

1. 导入不是文本粘贴

```
import sys
import json

assert sys.modules["json"] is json
```

普通导入后，模块对象进入缓存；后续导入通常复用它。导入语句还会在当前命名空间绑定名称。`from module import value` 绑定的是导入时获得的对象，模块后续重新绑定同名属性不会自动更新这个局部名称。

2. 包和入口

推荐把可导入业务代码放在包中，把命令行入口保持很薄：

```
project/
  pyproject.toml
  src/
    risk_service/
      __init__.py
      app.py
      domain.py
      repository.py
  tests/
```

`src` 布局能减少测试误从仓库根目录导入未安装代码的机会。运行模块可使用 `python -m risk_service...`，相对导入和包上下文比直接执行包内文件更稳定。

3. 循环导入

假设 A 导入 B，B 在 A 尚未执行完成时再次读取 A 的名称，看到的是“部分初始化模块”。失败点取决于名称何时定义，所以循环导入可能因代码顺序改变而出现或消失。

解决思路按优先级通常是：

- 抽取共同抽象到更底层模块。
- 让依赖方向单向。
- 把运行时注册改为显式组装。
- 仅在确有必要时局部导入，且说明原因。

局部导入能延迟问题，不一定修复架构依赖环。

> 深挖层：importlib
> 导入系统围绕 finder、loader、module spec 和缓存协作，并支持元路径钩子、命名空间包等扩展。框架插件系统可能利用 entry points 或自定义发现机制。业务代码不要直接篡改 `sys.path` 解决打包问题；源码和文档入口是 `importlib`。

4. 虚拟环境与依赖

虚拟环境隔离解释器可见的安装包，不等于生成完全可重现构建。可重现还需要：

- 明确支持的 Python 版本。
- 直接依赖和间接依赖的锁定策略。
- 构建工具、系统库和平台信息。

- 安全更新与兼容性验证。

`pyproject.toml` 可以声明构建系统、项目元数据、依赖、命令入口以及多个工具配置。不要同时维护多套互相冲突的依赖真相。

5. 配置边界

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Settings:
    database_url: str
    query_timeout_seconds: float
```

环境变量是输入渠道，不应在任意业务函数中随时读取。应用入口读取、校验并构造 Settings，再显式传递。测试可以传入隔离配置，避免环境污染。

> 边界案例：名称遮蔽
> 项目根目录若存在 `json.py`、`typing.py` 等文件，可能遮蔽标准库或第三方包。排查导入异常时先打印 `module.__file__`、`sys.path` 和当前工作目录，而不是立即重装依赖。

6. 项目与面试

FastMCP 服务可把协议适配、工具声明、领域服务、SQL 仓库和飞书通知拆开。工具模块不应在导入时连接数据库；应用组装层负责创建连接池并注册生命周期。

问：**`__init__.py`** 有什么作用？

它可以标记普通包、执行包初始化和控制导出；现代 Python 也支持没有它的命名空间包。是否使用取决于包分发和组织需求，不能只回答“没有就不能导入”。

> 练习
>
> 1. 建立一个最小 `src` 布局项目，并从干净虚拟环境安装测试。
> 2. 复现部分初始化导致的循环导入错误，再重构依赖方向。
> 3. 记录某模块的 `__file__`、`__spec__` 和 `sys.modules` 条目。

官方参考：The import system；Writing pyproject.toml

A20 类型注解与数据建模

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. 类型注解会在运行时自动拒绝错误类型吗？
2. `Any` 和 `object` 有什么关键区别？
3. Protocol 如何实现结构化类型？
4. 类型校验模型与领域对象为什么不一定是同一个类？

> 速记层
> 类型注解主要服务静态检查、IDE、文档和框架反射，Python 默认不强制运行时类型。`Any` 关闭相应检查，`object` 表示未知对象但使用前仍需缩小类型。边界输入要做运行时校验，内部用精确类型表达不变量。

1. 注解不是运行时防线

```
def double(value: int) -> int:
    return value * 2


assert double("a") == "aa"
```

解释器默认不会根据注解拒绝字符串。静态检查器应在运行前报告问题，但不可信 HTTP、数据库和 MCP 输入仍需运行时校验。

2. 联合、缩小与 None

```
def normalize_user_id(value: int | str | None) -> int | None:
    if value is None:
        return None
    if isinstance(value, int):
        return value
    return int(value)
```

类型缩小应和真实校验一致。仅使用 `cast()` 告诉检查器“相信我”，不会转换或验证运行时对象。

3. `Any` 与 `object`

```
from typing import Any


def unchecked(value: Any) -> None:
    value.not_existing()  # 静态检查通常放行


def unknown(value: object) -> None:
    if isinstance(value, str):
        print(value.upper())
```

`Any` 会向调用链传播不安全，适合逐步迁移或真正动态边界，但应尽快收窄。`object` 表示可以接收任何对象，同时要求使用前检查。

4. 泛型与 Protocol

```
from typing import Protocol, TypeVar

T = TypeVar("T")


class Repository(Protocol[T]):
    def get(self, key: str) -> T | None: ...
```

泛型让容器和接口保留元素类型。协变、逆变与不变描述类型参数在子类型替换中的方向；普通业务先从不变设计起步，只有读取者或写入者接口确实需要时再显式设计方差。

5. 数据边界与领域模型

来自外部的请求模型常需要：字段缺失处理、字符串转换、长度和枚举校验、错误定位。领域对象则强调业务不变量和行为。两者可以分开：

```
@dataclass(frozen=True)
class QueryWindow:
    days: int

    def __post_init__(self):
        if not 1 <= self.days <= 90:
            raise ValueError("days out of range")
```

边界校验框架负责把原始输入转换为 QueryWindow；领域层不必依赖 Web 或 MCP 框架。

> 深挖层：运行时注解
> 注解可保存在 `__annotations__`，但前向引用和延迟求值策略随版本演进。框架读取注解应使用 `typing.get_type_hints()` 等公开 API，并准备处理导入上下文和失败。不要手工假设注解一定是已解析类型对象。

6. TypedDict、Literal 与 NewType

TypedDict 描述字典形状，适合已有映射协议；Literal 限制字面值；NewType 在静态层区分底层类型相同的业务标识。它们不自动进行运行时校验。

> 项目映射
> MCP Tool 的注解和描述会影响自动生成 Schema。应避免 `dict[str, Any]` 作为所有输入输出；为查询条件、统计结果和错误结构定义明确模型，工具更容易被 Agent 正确调用，也更容易做兼容性测试。

7. 面试与练习

问：用了类型注解还需要测试吗？

需要。类型检查覆盖静态可表达的形状，无法证明业务规则、I/O 行为、并发、权限和性能正确；运行时动态数据也需要校验。

> 练习
>
> 1. 把一个 `dict[str, Any]` 查询结果改为 TypedDict 或 dataclass。

> 2. 为 Repository 定义 Protocol，并运行静态检查。
> 3. 找出项目中传播最远的 Any，逐层收窄。

官方参考：typing；Python typing specification

A21 多线程、多进程与 GIL

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. GIL 是否保证业务数据线程安全？
2. I/O 密集任务为什么可能从线程中获益？
3. 多进程为什么不能直接共享普通 Python 对象？
4. Free-threaded CPython 对传统结论有什么影响？

> 速记层
> 传统 CPython 构建中的 GIL 限制同一解释器内同时执行 Python 字节码，但不保证复合业务操作安全；阻塞 I/O 和部分原生扩展会释放 GIL。线程共享内存，进程拥有独立地址空间。Python 3.13 起提供可选 free-threaded 构建，应单独验证依赖、线程安全和性能。

1. 线程适合什么

```
from concurrent.futures import ThreadPoolExecutor


def fetch(url: str) -> bytes:
    ...


with ThreadPoolExecutor(max_workers=8) as pool:
    results = list(pool.map(fetch, urls))
```

网络、文件和数据库等待期间，线程可让其他任务推进。线程仍会消耗栈、连接和调度资源，Worker 数不能无限扩大。必须为下游连接池和接口限流设置同一容量预算。

2. GIL 不是数据锁

```
if key not in cache:
    cache[key] = build_value()
```

这包含检查、计算和写入多个步骤。线程可以在步骤间切换，两个线程可能重复计算或覆盖。即使某个内建操作在当前 CPython 看似原子，也不应把实现偶然性当成跨版本业务保证。

```
from threading import Lock

lock = Lock()
with lock:
    if key not in cache:
        cache[key] = build_value()
```

锁范围应覆盖需要保持不变量的完整事务，同时避免把慢 I/O 放在全局锁内。

3. 进程与序列化边界

```
from concurrent.futures import ProcessPoolExecutor
```

```
def cpu_work(value: int) -> int:
    return sum(i * i for i in range(value))


with ProcessPoolExecutor() as pool:
    results = list(pool.map(cpu_work, inputs))
```

多进程可以并行执行 CPU 密集 Python 代码，但参数和结果通常需要序列化，启动方式在不同平台不同。模块顶层副作用、无法 pickle 的闭包、巨型数据复制和进程崩溃都会影响设计。

4. 锁、队列与所有权

共享可变状态越少，正确性越容易证明。常见做法是：

- 一个 Worker 独占一个浏览器上下文或临时目录。
- 通过线程安全队列传递任务和结果。
- 使用不可变消息而不是共享大型字典。
- 给锁规定固定获取顺序，降低死锁风险。

超时不能自动中止正在执行的线程函数；Future 超时常只是调用者停止等待。需要强制隔离不可信或可能卡死的工作时，进程边界更容易终止，但还要清理子进程和外部资源。

> 深挖层：Free-threaded CPython
> Python 3.13 起官方安装方式可选 free-threaded 构建，允许同一解释器内多个线程同时执行 Python 代码。它不是传统构建自动取消 GIL，也不让旧代码自动线程安全。C 扩展兼容、对象保护、容器并发语义和单线程性能都要按实际版本验证。Python 3.11 项目仍以传统 GIL 模型为基线。

5. 项目选型

pytest-xdist 使用多个进程运行测试，主要收益是隔离和并行；每个 Worker 必须拥有独立账号、数据库命名空间、端口、下载目录和浏览器状态。仅把 `-n 5` 打开而不设计资源隔离，会把顺序依赖变成 Flaky。

> 边界案例
> NumPy、压缩、加密等原生库可能在计算时释放 GIL，线程可获得 CPU 并行收益；也可能不释放。只能依据库文档和基准测试，不能按“CPU 密集必用进程”机械判断。

6. 面试与练习

问：有 GIL 为什么还需要锁？

GIL保护解释器内部执行，不保护你的多步骤业务不变量；I/O 和 C 扩展还可能释放 GIL。锁是为共享状态的一致性设计。

> 练习
>
> 1. 写一个并发缓存初始化测试，复现重复构造，再用锁修复。
> 2. 分别用线程池和进程池运行 I/O 与 CPU 基准，记录启动和序列化成本。
> 3. 为 xdist Worker 设计独立资源命名规则。

官方参考：threading；multiprocessing；Free-threading HOWTO

A22 asyncio 异步编程

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. 调用 `async def` 会立即执行函数体吗？
2. 协程、Task 和 Future 有什么关系？
3. 为什么同步数据库或 HTTP 调用会阻塞整个事件循环？
4. 取消为什么不是“立刻杀死任务”？

> 速记层
> 调用异步函数得到协程对象，`await` 或 Task 调度后才执行。事件循环在任务主动让出控制权时切换；任何长时间同步阻塞都会卡住同一循环上的其他任务。可靠异步代码必须处理超时、取消、并发上限、异常汇总和资源清理。

1. 协程与 Task

```
import asyncio


async def fetch_one(value: int) -> int:
    await asyncio.sleep(0.01)
    return value * 2


async def main():
    coroutine = fetch_one(2)
    task = asyncio.create_task(coroutine)
    result = await task
    assert result == 4


asyncio.run(main())
```

协程对象描述可暂停计算；Task 把协程纳入事件循环调度并保存完成状态。创建协程后忘记 await，通常会产生警告且工作没有执行。

2. 并发不等于无限并发

```
async def run_limited(items, limit: int):
    semaphore = asyncio.Semaphore(limit)

    async def one(item):
        async with semaphore:
            return await fetch_one(item)

    return await asyncio.gather(*(one(item) for item in items))
```

即使任务只是 I/O，也会占用连接、内存和下游配额。并发上限应与数据库连接池、HTTP 服务限制和超时预算协调。一次为百万项创建百万 Task 仍会耗尽内存，可以使用固定 Worker 和队列形成背压。

3. 超时和取消

```
async def guarded():
```

```
    try:
        async with asyncio.timeout(2):
            return await remote_call()
    finally:
        await close_resource()
```

取消通过在可取消点向任务注入取消异常来协作完成。任务若吞掉取消、长期不 await 或卡在阻塞系统调用中，就不能及时停止。清理代码应在 finally 或异步上下文管理器中执行，通常清理后继续传播取消。

4. TaskGroup 与结构化并发

```
async def collect(ids):
    tasks = []
    async with asyncio.TaskGroup() as group:
        for user_id in ids:
            tasks.append(group.create_task(fetch_user(user_id)))
    return [task.result() for task in tasks]
```

TaskGroup 把子任务生命周期限制在上下文中，一个子任务失败时会取消其余任务，并以异常组报告失败。相比创建“后台任务”后丢失引用，它更容易保证退出时没有遗留任务。

5. 阻塞函数

```
async def call_blocking(argument):
    return await asyncio.to_thread(blocking_library_call, argument)
```

`to_thread` 可把阻塞调用移到线程，但不会让库本身变成可取消或线程安全。线程中的函数继续运行时，外层协程取消可能只停止等待。CPU 密集任务仍需评估进程或原生实现。

> 深挖层：事件循环与就绪队列
> 事件循环管理就绪回调、计时器和 I/O 通知；Task 每次驱动协程前进到下一个挂起点。公平性并非绝对保证，长回调会拖延所有任务。调试时可以启用 asyncio debug、记录慢回调、检查未关闭资源和悬挂任务。

6. 上下文传播

`contextvars` 适合请求 ID 等异步任务局部状态，比线程局部变量更能跟随协程上下文。创建 Task 时上下文会按规则复制；跨线程、进程和自定义执行器时要验证传播。

> 项目映射
> MCP 工具若定义为 async，却内部直接调用同步 MySQL 驱动或 `requests`，会阻塞事件循环。应选择异步驱动、受控线程池，或保持整个服务同步模型一致；还要把调用超时继续传递到数据库层。

7. 面试与练习

问：asyncio 比多线程快吗？

没有普遍结论。它在大量可等待 I/O、需要精确控制任务和连接数时有优势；同步库、CPU 工作和复杂取消会抵消收益。应按调用链选型。

> 练习

1. 写一个并发上限为 5 的异步抓取器，并记录最大同时执行数。
2. 构造阻塞 `time.sleep()` 卡住事件循环的例子，再用 `to_thread` 隔离。
3. 使用 TaskGroup 制造两个失败任务，检查 ExceptionGroup。

官方参考：asyncio；Coroutines and Tasks

第二篇：Python 后端与数据库

B01 HTTP、REST 与 Web 请求生命周期

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. HTTP 方法的“安全”和“幂等”分别是什么意思？
2. 401 与 403 应如何区分？
3. 请求超时后，服务端操作一定停止了吗？
4. REST 是否等于“URL 使用名词并返回 JSON”？

> 速记层
> HTTP 是请求 - 响应协议，方法、目标、Header、状态码和消息体共同表达语义。安全方法原则上不改变服务器状态；幂等表示重复相同请求的预期效果等同一次。网络超时只说明调用方没有按时拿到结果，不证明服务端没有执行。

1. 请求和响应

一个典型 API 请求包含：

- 方法：GET、POST、PUT、PATCH、DELETE 等。
- 目标：路径和查询参数。
- Header：内容类型、认证、条件请求、追踪信息。
- Body：JSON、表单、文件或其他媒体类型。

响应包含状态码、Header 和可选 Body。状态码表达协议层结果，响应体补充业务信息。不要始终返回 200，再把真实失败藏在 `{"code": 500}` 中；这会破坏代理、监控和客户端通用处理。

2. 常用方法语义

方法	安全	幂等	常见语义
GET	是	是	读取资源，不应产生业务副作用
POST	否	通常否	创建、提交命令或触发处理
PUT	否	是	用完整表示创建或替换目标资源
PATCH	否	不一定	部分修改，取决于补丁语义
DELETE	否	是	使目标资源处于已删除状态

幂等描述服务端预期效果，不表示每次响应必须完全相同，也不表示请求可以无限重试。日志、计费和通知等副作用仍需设计。

3. 状态码和错误体

- 400：请求语法或通用校验失败。
- 401：缺少或无效认证凭据，通常意味着“尚未认证”。
- 403：身份已知但没有权限。
- 404：资源不存在；有时也用于隐藏资源是否存在。
- 409：当前资源状态冲突。
- 422：语法可解析但实体语义校验失败，是否使用取决于 API 规范。
- 429：请求过多。
- 500：未预期服务端错误。
- 502 / 503 / 504：网关、临时不可用或上游超时类问题。

错误体应包含稳定错误码、用户可理解消息、必要字段定位和请求 ID，不应泄露堆栈、SQL 或密钥。

4. 超时与不确定结果

```
response = client.post(
    url,
    json=payload,
    timeout=3,
    headers={"Idempotency-Key": operation_id},
)
```

客户端超时时，请求可能：尚未到达、服务端正在执行、已经成功但响应丢失，或已经失败。对会产生副作用的操作，盲目重试可能重复创建或重复通知。需要幂等键、状态查询或可恢复工作流。

> 深挖层：连接与消息边界
> HTTP 语义独立于具体传输版本，但 HTTP/1.1、HTTP/2 和 HTTP/3 在连接复用、并发流和队头阻塞等方面不同。应用层仍需设置整体截止时间和上游超时，不能因为连接支持多路复用就忽略资源上限。

5. REST 的边界

REST 强调资源、统一接口、无状态交互和可缓存语义等约束。业务命令不一定都能自然建模为 CRUD；可以把命令建模成资源，例如创建一次审批动作。核心是协议清晰和状态可追踪，不是机械追求“URL 绝不能出现动词”。

6. 项目与面试

飞书机器人调用风险查询 API 是一条跨系统链路：飞书事件接收、鉴权、参数解析、数据库查询、结构化结果和卡片发送分别需要超时与错误边界。用户看到“查询超时”时，后台查询可能仍在运行，需要请求 ID 关联日志并避免重复通知。

问：GET 能不能带 Body？

不同组件支持差异大且缓存语义不清，通用 API 不应依赖 GET Body。复杂查询可使用明确查询参数、POST

查询资源或专门搜索端点。

> 练习
>
> 1. 为创建任务接口设计成功、重复提交、参数错误和权限不足的状态码与错误体。
> 2. 画出客户端超时后四种可能状态，并设计查询或幂等恢复。
> 3. 检查一个现有接口的方法语义是否与重试策略一致。

官方参考：RFC 9110 HTTP Semantics

B02 Flask 项目架构与请求生命周期

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. Application Factory 解决什么问题？
2. `current_app`、`request` 和 `g` 为什么不需要层层传参？
3. `g` 能否用来保存跨请求数据？
4. Flask 请求上下文和应用上下文按什么顺序退出？

> 速记层
> 应用工厂延迟创建 Flask 实例，使配置、扩展和测试环境可组合。WSGI 服务器调用 Flask 应用，框架建立应用上下文和请求上下文，匹配路由、执行视图、生成响应，再依次清理请求和应用资源。上下文本地代理不是普通全局变量。

1. 应用工厂

```
from flask import Flask


def create_app(config: dict | None = None) -> Flask:
    app = Flask(__name__)
    app.config.from_mapping(
        QUERY_TIMEOUT_SECONDS=3,
    )
    if config:
        app.config.from_mapping(config)

    from .api import api
    app.register_blueprint(api, url_prefix="/api")
    return app
```

工厂允许测试创建多个配置不同的应用，减少模块导入时连接外部资源，也避免其他模块直接导入单例 app 造成循环依赖。

扩展通常先创建未绑定对象，再在工厂中 `init_app(app)`。连接池、MCP Client 等有生命周期的资源应明确何时创建和关闭。

2. Blueprint 是注册蓝图

Blueprint 记录路由和处理器等注册动作，本身不是独立应用。它适合按业务能力组织 API：

```
from flask import Blueprint, jsonify

api = Blueprint("risk", __name__)


@api.get("/health")
def health():
    return jsonify(status="ok")
```

不要把所有业务逻辑写进视图函数。视图负责协议适配，领域服务负责业务规则，仓库负责数据访问。

3. 生命周期

典型流程可以概括为：

1. WSGI Server 调用 Flask 应用。
2. Flask 根据 WSGI environ 创建 RequestContext。
3. 推入应用上下文，使 `current_app` 和 `g` 可用。
4. 推入请求上下文，使 `request` 和 `session` 可用。
5. 执行 before_request、路由匹配和视图。
6. 把返回值转换为 Response，执行 after_request。
7. 弹出请求上下文并执行 teardown_request。
8. 弹出应用上下文并执行 teardown_appcontext。

teardown 会在异常路径执行，因此适合关闭连接；它不表示请求成功，不能无条件提交事务。

4. 上下文本地代理

```
from flask import current_app, g


def get_repository():
    if "repository" not in g:
        g.repository = build_repository(current_app.config)
    return g.repository
```

`current_app`、`request`、`g` 是代理，根据当前上下文定位真实对象。`g` 的生命周期通常与应用上下文一致，在 Web 请求中通常只持续一个请求，不适合跨请求缓存用户数据。

Flask 现代版本使用 Python context vars 管理上下文。任务切换、后台线程或异步边界不能假设上下文自动存在；后台任务应显式传递所需数据，而不是把整个 request 对象带走。

> 深挖层：WSGI 调用链
> Flask 实例是 WSGI callable，通常通过 `__call__` 进入 `wsgi_app()`。异常处理、上下文 push/pop 和响应迭代都在这条链路中。阅读官方 lifecycle 文档后，再从 `flask/app.py` 的 `wsgi_app`、`full_dispatch_request` 等入口跟源码，比从路由装饰器向下盲查更有效。

5. 错误处理与 teardown

```
@app.errorhandler(DomainError)
def handle_domain_error(exc):
    return {"code": exc.code, "message": str(exc)}, 400
```

错误处理器负责领域异常到 HTTP 的映射；未预期异常应记录 request ID 并返回通用 500。teardown 函数用于清理，不应覆盖原异常或在清理失败时丢失根因。

> 边界案例：开发服务器
> Flask 自带开发服务器和 Debugger 面向本地开发，不是生产部署方案。生产应使用合适 WSGI Server、反向代理、超时和 Worker 模型；绝不能把可交互调试器暴露到不可信网络。

6. 项目与面试

提审服工具可以由 Flask 管理健康检查和内部 API，由独立 FastMCP 适配层复用同一领域服务。共享的是服务对象和仓库接口，不是复制 SQL 或让 Flask 视图直接被 MCP 调用。

问：为什么会出现 Working outside of application context？

代码在没有活动应用上下文时访问 `current_app` 或依赖它的对象。应把逻辑放进正确请求/CLI 生命周期，测试中显式创建上下文，或更好地把配置作为参数传入纯业务代码。

> 练习
>
> 1. 创建两个测试配置不同的 Flask 应用，证明状态不互相污染。
> 2. 记录 before、view、after、teardown 的执行顺序，包括异常路径。
> 3. 把视图中的 SQL 拆到服务和仓库，并用内存仓库单测服务。

官方参考：Flask lifecycle；Application context；Application factories

B03 API 参数校验、序列化与错误处理

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. 类型转换和业务校验有什么区别？
2. “字段未提供”和“字段值为 null”为什么要区分？
3. 序列化模型能否直接替代数据库模型和领域模型？
4. 错误码为什么应稳定而消息可以调整？

> 速记层
> 所有外部输入都要经过解析、类型转换、结构校验和业务约束；校验通过后再构造内部模型。输出序列化需要稳定 Schema、时间和数字规则，不能直接泄露 ORM 对象。错误响应分离机器可处理代码、用户消息、字段定位和追踪 ID。

1. 分层校验

```
from dataclasses import dataclass


@dataclass(frozen=True)
class QueryRequest:
    package_id: str
```

```
    days: int


def parse_request(data: dict) -> QueryRequest:
    package_id = str(data["package_id"]).strip()
    days = int(data.get("days", 7))
    if not package_id:
        raise ValidationError("package_id is empty")
    if not 1 <= days <= 90:
        raise ValidationError("days must be between 1 and 90")
    return QueryRequest(package_id, days)
```

现实项目通常使用成熟校验库，但仍要理解四层：

1. 能否解析媒体类型和 JSON。
2. 字段是否存在、类型能否转换。
3. 单字段范围和格式是否有效。
4. 多字段组合和业务状态是否允许。

数据库唯一约束、权限和当前状态往往只能在服务层验证，不能全塞进请求模型。

2. 缺失、空值和空字符串

PATCH 接口中三种状态可能不同：

- 没有 `name` 字段：保持原值。
- `name: null`：明确清空，若业务允许。
- `name: ""`：传入空字符串，可能非法。

使用哨兵或校验库的 fields-set 信息保留区别。把它们都 `dict.get()` 成 None 会丢失语义。

3. 序列化边界

时间必须包含时区或明确约定 UTC；Decimal 要决定输出字符串还是 JSON number；大整数在 JavaScript 客户端可能超出安全精度；枚举要稳定；内部字段和密钥不可直接导出。

```
from datetime import datetime, timezone

timestamp = datetime.now(timezone.utc).isoformat()
```

不要直接 `json.dumps(obj.__dict__)`。它会把内部结构当成公共契约，字段重构可能无意破坏客户端。

4. 统一错误模型

```
{
  "code": "INVALID_QUERY_WINDOW",
  "message": "days must be between 1 and 90",
  "field": "days",
  "request_id": "req_123"
}
```

稳定 `code` 供客户端分支；`message` 可以本地化或优化。错误体不应返回 SQL、绝对路径、堆栈或下游密钥。日志通过 request_id 关联内部详细异常。

> 深挖层：Schema 演进

> 新增可选字段通常比删除或改变字段类型更容易兼容。默认值在服务端和客户端可能不一致；把“缺失即默认”写入契约。对 MCP Tool，Schema 改名可能使已有 Prompt 和 Agent 规划失效，因此需要版本测试和兼容窗口。

5. 批量错误与快速失败

用户表单可能适合一次返回多个字段错误；涉及权限、资源消耗或安全边界时可以快速失败。批量导入应区分整个请求原子失败与逐行结果，避免部分成功后客户端误以为全部回滚。

> 项目映射
> 风险工具的自然语言参数最终必须收敛为结构化 QueryRequest：包 ID、日期范围、内外网选项和 TopN 范围均需校验。模型输出不能绕过这一层直接拼接 SQL。

6. 面试与练习

问：为什么不直接把 ORM 模型返回 JSON？

数据库模型表达持久化结构，API 模型表达外部契约；直接暴露会泄漏字段、触发懒加载、形成循环引用，并把数据库重构传播给客户端。

> 练习
>
> 1. 设计 PATCH 模型，区分缺失、null 和空字符串。
> 2. 为 Decimal、时区时间和大整数定义 JSON 规则并写契约测试。
> 3. 将一个数据库异常映射为稳定错误码，同时保留内部原因链。

官方参考：Flask error handling；RFC 9457 Problem Details

B04 认证、授权与接口安全

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. 认证与授权分别回答什么问题？
2. JWT 为什么不是“加密后的用户信息”？
3. Session 和 JWT 的撤销、轮换与泄露处理有什么差异？
4. 只在前端隐藏按钮是否构成权限控制？

> 速记层
> 认证确认主体身份，授权判断主体能否执行动作。服务端必须在资源和操作层执行授权；前端控制只改善体验。Session、JWT 和 API Key 都是凭据机制，不自动解决权限、撤销、重放、密钥轮换和审计。

1. 密码与凭据

密码不能明文保存，也不应使用普通快速哈希直接存储。应采用专门密码哈希算法、随机盐和合适成本参

数，并规划算法升级。登录接口需要速率限制、通用错误消息和审计，避免账号枚举。

凭据不得出现在 URL、异常、普通日志和前端存储的非必要位置。TLS 保护传输，不保护终端被入侵或日志泄露。

2. Session 与 Token

服务端 Session 常让客户端保存随机标识，状态主要在服务端；JWT 把声明封装在签名 Token 中，服务端验证签名和标准声明。JWT 通常只是编码加签，不是保密加密，Payload 可能被直接读取。

验证 JWT 至少要固定允许算法并检查：

- 签名和密钥来源。
- `exp`、`nbf` 等时间约束。
- `iss` 和 `aud`。
- Token 类型和用途。
- 用户、设备或权限是否仍有效。

短期访问令牌配合受控刷新令牌、轮换和撤销策略比超长有效期令牌更安全，但系统复杂度更高。

3. 授权必须绑定资源

```
def can_view_report(actor, report) -> bool:
    return (
        actor.is_admin
        or report.owner_id == actor.user_id
        or actor.team_id == report.team_id
    )
```

只有角色判断而没有资源归属检查，会产生 IDOR / 越权访问。数据库查询最好同时包含授权条件，而不是先按任意 ID 取出再遗漏检查。

4. Cookie、CSRF 与 CORS

Cookie 可以设置 Secure、HttpOnly、SameSite 等属性。浏览器会自动携带 Cookie，因此改变状态的 Cookie 认证请求需要考虑 CSRF。CSRF Token、SameSite 和来源检查是互补措施。

CORS 控制浏览器脚本能否读取跨源响应，不是服务端认证，也不会阻止 curl 或后端程序请求接口。宽泛允许 Origin 与凭据组合会造成风险。

> 深挖层：权限缓存
> 把权限放入 Token 或缓存可减少数据库查询，但权限撤销会有延迟。必须明确“最多多久生效”、高风险动作是否实时查询、缓存键是否包含租户和版本，以及密钥轮换时旧 Token 如何处理。

5. 内网不是免安全理由

VPC、内部群机器人和服务账号降低部分暴露面，但仍需：

- 最小数据库权限和只读账号。
- 工具级允许列表。

- 调用主体和查询参数审计。
- 返回数据脱敏和行数限制。
- 超时、并发和频率限制。

> 项目映射
> 提审服工具即使运行在内网，也应把“谁能调用什么查询”与“MySQL 凭据能访问什么表”分成两层控制。飞书群成员身份不能自动等价为数据库权限。

6. 面试与练习

问：JWT 和 Session 哪个更安全？

没有绝对答案。安全取决于存储、传输、生命周期、撤销、密钥管理和实现质量。JWT 便于分布式验证，但泄露后撤销和权限同步更复杂；Session 需要服务端状态和扩展方案。

> 练习
>
> 1. 为“查看自己团队报告”和“管理员查看全部”写授权测试。
> 2. 列出访问令牌泄露后的检测、撤销和密钥轮换步骤。
> 3. 检查一个 Cookie 认证接口的 CSRF 防护和 CORS 配置。

官方参考：Flask Web Security；RFC 7519 JWT；OWASP ASVS

B05 Python 连接 MySQL

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. 数据库连接池解决什么问题，又会引入什么风险？
2. 参数化查询为什么不同于字符串转义？
3. 游标取出全部结果和流式读取如何取舍？
4. 请求结束时应该关闭连接、归还连接，还是提交事务？

> 速记层
> Python 驱动把调用转换为数据库协议。连接昂贵且有上限，连接池复用连接，但必须验证健康、设置等待超时和正确归还。SQL 值使用驱动参数绑定；表名、列名等结构不能当普通值参数，需要白名单构造。

1. 连接生命周期

```
def fetch_user(connection, user_id: int):
    with connection.cursor() as cursor:
        cursor.execute(
            "SELECT user_id, region FROM users WHERE user_id = %s",
            (user_id,),
        )
        return cursor.fetchone()
```

参数占位符样式取决于驱动，不要把上例机械复制到所有库。连接应通过上下文或 try/finally 归还；关闭池中代理通常表示归还，不一定关闭底层 TCP。

2. 参数化查询

错误写法：

```
sql = f"SELECT * FROM users WHERE name = '{name}'"
```

正确参数化让驱动按协议传递 SQL 与值，数据库区分结构和数据。手工替换引号不能覆盖字符集、注释、反斜杠和不同 SQL 模式。

结构标识符通常不能使用值占位符：

```
ALLOWED_SORT = {
    "created_at": "created_at",
    "score": "risk_score",
}
column = ALLOWED_SORT[user_sort]
sql = f"SELECT ... ORDER BY {column} DESC"
```

映射值必须来自代码白名单，而不是把用户输入原样插入。

3. 连接池容量

Web Worker 数乘以每进程池大小，可能远超 MySQL 最大连接数。容量应共同预算：

```
总潜在连接 ≈ 实例数 × 进程数 × 每进程池上限
```

池需要连接获取超时、空闲回收、断线验证和失败指标。连接不可跨进程继承使用；fork 前创建的池可能让多个进程共享损坏的文件描述符状态。

4. 批量和流式读取

`fetchall()` 简单但可能占用大量内存；分批 `fetchmany()` 或服务端游标降低内存，却延长连接和事务占用。批量写入可减少往返，但要受包大小、锁持有时间和错误定位限制。

> 深挖层：DB-API 抽象边界
> Python DB-API 规定连接、游标、异常层级和参数风格等共同接口，但驱动在线程安全、自动提交、服务端游标、返回类型和连接池方面差异明显。生产代码必须锁定驱动并阅读其文档，不能只依赖“都符合 DB-API”。

5. 类型和时区

DECIMAL 应映射为 Decimal；DATETIME 是否带时区取决于字段和驱动；JSON、BIT、NULL 等也需验证。不要在数据访问层把所有值转成字符串再交给业务层。

> 边界案例：超时后的连接
> 查询超时或网络断开后，连接可能仍有未读取响应或未知事务状态，不能无条件放回池继续复用。驱动和池需要决定回滚、丢弃或重建连接。

6. 项目与面试

风险查询会关联多表并进行聚合。应限制时间范围和最大行数，为查询设置超时，记录模板 ID 和耗时而不是完整敏感 SQL 参数。只读工具使用最小权限账号。

问：连接池越大吞吐越高吗？

不是。过大连接数增加数据库上下文切换、内存和锁竞争，并可能压垮实例。应从数据库容量、查询时延和服务并发联合压测。

> 练习
>
> 1. 计算三实例、四进程、每进程十连接时的最大连接预算。
> 2. 为动态排序字段实现白名单，并测试注入输入。
> 3. 模拟查询异常，验证连接回滚并正确归还或丢弃。

官方参考：PEP 249 Python DB-API；MySQL Connector/Python

B06 事务、隔离级别与数据库锁

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. ACID 中的一致性是否等于“多个副本立即一致”？
2. MVCC 是否意味着所有读都不加锁？
3. 死锁是不是数据库 Bug？遇到后是否应重试？
4. 为什么事务不能跨网络调用保持很久？

> 速记层
> 事务把一组数据库操作纳入原子提交或回滚边界。隔离级别决定并发读写可见性和锁行为；InnoDB 结合多版本并发控制与锁。死锁是并发锁系统可预期结果，数据库会回滚参与者之一，应用需保持事务可重试且副作用可控。

1. ACID 的工程含义

- Atomicity：事务内数据库修改全部提交或全部回滚。
- Consistency：事务把数据库从满足约束的状态带到另一个满足约束的状态；业务不变量仍需正确代码和约束共同维护。
- Isolation：并发事务按隔离规则观察彼此。
- Durability：提交后的结果在承诺的故障模型下持久保存。

ACID 不保证外部 HTTP、飞书消息和文件写入跟数据库自动原子一致。

2. 事务边界

```
def transfer(connection, sender: int, receiver: int, amount):
    try:
```

```
        debit(connection, sender, amount)
        credit(connection, receiver, amount)
    except Exception:
        connection.rollback()
        raise
    else:
        connection.commit()
```

生产代码还需锁定行、检查余额、处理并发和精度。不要在事务内等待用户输入、调用慢外部 API 或进行长计算；这会长时间占用连接、行版本和锁。

3. 隔离现象

常见讨论包括：

- 脏读：读到未提交修改。
- 不可重复读：同一事务两次读取同一行得到不同已提交结果。
- 幻读：相同条件查询得到不同记录集合。
- 丢失更新：两个参与者基于旧值写回，覆盖彼此。

只背隔离级别表格不够。MySQL InnoDB 的默认隔离、快照创建、当前读、锁定读和索引范围会影响实际行为，应通过两个连接复现实验。

4. MVCC 与锁定读

普通一致性读取可以从版本链构造快照，通常不加普通记录锁；更新、删除、`SELECT ... FOR UPDATE` 等当前读需要锁。没有合适索引时，扫描和锁范围可能扩大。

```
SELECT status
FROM jobs
WHERE job_id = ?
FOR UPDATE;
```

锁定后检查并修改，防止多个 Worker 同时领取同一任务。更高吞吐场景还可使用条件更新或 `SKIP LOCKED`，但必须定义饥饿和失败恢复。

5. 死锁

两个事务以不同顺序获取资源可能形成环：

```
T1 锁住 A，等待 B
T2 锁住 B，等待 A
```

降低死锁的方法：固定访问顺序、缩短事务、建立合适索引、减少锁范围。数据库检测死锁后会回滚一个事务；应用只对可重试事务进行有限、带抖动重试，并记录根因。

> 深挖层：undo 与 Read View
> InnoDB 在 undo 相关结构中保留旧版本信息，快照读取根据事务可见性规则选择版本。长事务会让旧版本不能及时清理，增加存储和查询成本。理解 MVCC 要同时看版本、可见性、当前读和锁，而不是简单记成“读写不冲突”。

6. 数据库与外部副作用

事务提交后再发送飞书消息，进程可能在二者之间崩溃；先发消息再提交，数据库又可能回滚。常见方案是事务内写 outbox 记录，提交后由独立 Worker 可靠投递，并以幂等键去重。

> 项目映射
> 自动化平台领取任务可以用条件更新：只有状态仍为 pending 时改为 running，并检查受影响行数。这样把“检查 + 更新”交给数据库原子执行，避免先查后改竞争。

7. 面试与练习

问：死锁后直接重试就行吗？

只有事务本身幂等、外部副作用未重复、重试次数受限且根因可观测时才安全。持续死锁还需修复锁顺序或索引。

> 练习
>
> 1. 用两个连接复现不可重复读或锁等待，并记录隔离级别。
> 2. 构造相反更新顺序触发死锁，再统一顺序修复。
> 3. 设计数据库写入加飞书通知的 outbox 流程。

官方参考：InnoDB transaction model；InnoDB multi-versioning；Deadlocks

B07 索引、执行计划与查询优化

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. InnoDB 聚簇索引和二级索引分别保存什么？
2. 联合索引的列顺序如何影响等值、范围和排序？
3. EXPLAIN 是实际执行结果还是优化器估计？
4. 为什么“加索引”可能让写入和锁竞争更糟？

> 速记层
> InnoDB 聚簇索引叶子保存行数据，二级索引叶子通常保存索引列和主键；二级索引查询可能回表。联合索引服务特定查询形状，选择性、最左前缀、范围、排序和覆盖共同决定效果。优化从正确 SQL、实际计划和真实数据分布出发。

1. 聚簇与二级索引

主键通常形成聚簇索引，叶子页保存完整行。二级索引叶子包含二级键和主键值，查找非覆盖字段时再通过主键访问聚簇索引。

因此主键过宽会膨胀所有二级索引；随机主键可能增加页分裂和局部性问题，但不能只凭一句“自增主键最好”忽略分布式生成、业务暴露和写热点等约束。

2. 联合索引

```
CREATE INDEX idx_login_package_time
ON login_events(package_id, created_at, user_id);
```

它适合以 package_id 等值过滤并按时间范围查询的形状。若只有 `created_at` 条件，是否有效取决于优化器和其他机制，不能期待普通最左前缀跳过第一列。

列顺序要结合：

- 等值条件和范围条件。
- 排序 / 分组需求。
- 选择性和数据分布。
- 覆盖字段与索引宽度。
- 写入频率和维护成本。

经验规则只是候选，最终用实际查询和数据验证。

3. 覆盖索引与回表

查询所需列均能从索引获得时，可以减少回表。但把大量字段塞进索引会增加磁盘、缓存和写放大。覆盖应针对高价值热点查询，不是把表复制进索引。

4. EXPLAIN

EXPLAIN 展示优化器选择和估计，包括访问方法、候选 / 实际索引、预计行数和附加操作。估计受统计信息影响，可能与真实执行偏差很大。支持实际执行分析的功能会提供更接近运行时的数据，但会真的执行查询，生产使用必须评估副作用和成本。

优化步骤：

1. 先确认结果语义正确。
2. 捕获完整 SQL 形状、参数范围和耗时分布。
3. 查看执行计划和扫描行数。
4. 检查索引、数据分布、隐式转换和函数包裹列。
5. 用真实规模压测并观察写入影响。

5. 常见失效或低效原因

- 类型不一致导致隐式转换。
- 对索引列施加无法利用索引的函数。
- 前导通配符模糊匹配。
- OR 条件和复杂表达式使候选成本变化。
- 返回过多列、过多行。
- 统计信息过旧或数据严重倾斜。
- 深分页扫描并丢弃大量行。

> 深挖层：优化器是成本模型
> 数据库不是看到索引就必用。优化器估算不同访问路径的 I/O、CPU 和行数，可能认为全表扫描更便

宜。强制索引会把当前数据分布的判断写死，应先修复统计、SQL 或索引设计，并持续观察计划漂移。

6. Python 侧优化边界

N+1 查询常由循环逐条访问数据库造成：

```
for user_id in user_ids:
    rows.append(repository.get_user(user_id))
```

可以批量查询、Join 或预加载，但批量 IN 也有参数和计划边界。数据库往返通常比 Python 字典查找昂贵，先减少往返再考虑微优化循环。

> 项目映射
> 风险工具按包体、日期、内外网和地区聚合。索引要围绕高频过滤和 Join 键设计；TopN 与分位数若从大结果集搬到 Python，网络和内存会成为瓶颈。应比较数据库聚合与 Python 后处理的准确性、资源和可维护性。

7. 面试与练习

问：联合索引把选择性最高的列放最前一定最好吗？

不一定。还要看查询前缀、范围、排序、覆盖、写入和数据分布。选择性是因素之一，不是唯一公式。

> 练习
>
> 1. 为按 package_id 和时间范围查询设计两个候选索引并比较计划。
> 2. 复现 N+1 查询，改成批量访问并记录数据库往返数。
> 3. 找出深分页查询，改为基于稳定游标的分页。

官方参考：Clustered and secondary indexes；EXPLAIN

B08 缓存、分页、幂等、重试与限流

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. 缓存一致性问题的根源是什么？
2. Offset 深分页为什么越来越慢并可能重复 / 漏数据？
3. 幂等键应绑定哪些请求信息？
4. 重试、超时和限流为什么必须联合设计？

> 速记层
> 缓存以复杂的一致性和失效换取时延或容量收益；分页必须有稳定唯一顺序；幂等让重复请求得到同一业务效果；重试只适用于瞬时且可重试失败；限流保护有限资源。五者都依赖明确状态和时间窗口。

1. Cache-Aside

```
def get_report(key):
    cached = cache.get(key)
    if cached is not None:
        return cached
    value = repository.load(key)
    cache.set(key, value, ttl=60)
    return value
```

需要继续回答：缓存穿透、击穿、雪崩、并发回源、负缓存、TTL 抖动和旧值容忍。缓存键必须包含租户、权限、查询版本和全部影响结果的参数，否则会串数据。

写路径常见“先写数据库，再删除缓存”，但两步间仍有失败窗口。应根据业务容忍度使用重试、消息通知、版本号或短 TTL，并监控不一致。

2. 分页

Offset 分页简单：

```
SELECT ...
FROM events
ORDER BY created_at DESC, event_id DESC
LIMIT 50 OFFSET 10000;
```

数据库可能仍需扫描并丢弃前面大量行；并发插入会使跨页结果移动。游标分页记录上页最后的稳定排序键：

```
WHERE (created_at, event_id) < (?, ?)
ORDER BY created_at DESC, event_id DESC
LIMIT 50;
```

排序必须唯一且方向一致，游标应签名或校验，不能信任客户端任意注入查询条件。

3. 幂等键

服务端可将（主体，操作，幂等键）关联请求摘要、状态和结果。相同键但请求体不同应返回冲突，而不是复用错误结果。记录需有合理过期、并发唯一约束和“处理中”恢复策略。

数据库唯一键是实现幂等的重要基础：先抢占操作记录，再执行业务；但外部副作用仍需 outbox 或对方幂等能力。

4. 超时预算和重试

一次用户请求总预算 5 秒时，不能让三个下游各自默认等待 5 秒再重试三次。应从总截止时间向下分配连接、读取和处理预算，并保留响应时间。

重试策略包括：

- 只重试明确瞬时错误。
- 指数退避加随机抖动。
- 有限次数和总截止时间。
- 操作幂等或带幂等键。
- 记录每次尝试及最终结果。

> 深挖层：重试风暴
> 下游变慢时，上游立即重试会成倍增加负载，使恢复更困难。限流、并发舱壁、熔断、退避和容量降级必须组合；熔断器自身也有半开探测和误判问题，不能当万能组件。

5. 限流

固定窗口简单但边界突发明显；滑动窗口更精确但成本高；令牌桶允许受控突发并限制长期速率。限流维度可按用户、租户、IP、工具和昂贵查询分别设置。

429 响应可以提示重试时间，但客户端仍需抖动，避免所有请求同时恢复。

> 项目映射
> 飞书群机器人可能在故障时被多人重复 @。应按群、用户和查询目标限制并发，复用短时间相同查询结果，并保证缓存键包含数据权限和时间范围。限流拒绝要给清楚反馈，而不是静默丢消息。

6. 面试与练习

问：Redis 挂了，服务应该挂吗？

取决于缓存是否只是加速还是承载会话、锁或幂等状态。需要预先定义降级：回源可能压垮数据库，直接失败可能更安全；不能临时拍脑袋。

> 练习
> 1. 设计包含租户、权限版本和查询参数的缓存键。
> 2. 实现稳定游标编码 / 解码并测试并列时间。
> 3. 画出超时 5 秒请求经过两个下游和两次重试的预算。

官方参考：HTTP semantics；Redis developer docs

B09 后端分层、后台任务与可靠性

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. Controller、Service、Repository 各自应该知道什么？
2. 把任务放进队列是否等于“保证执行一次”？
3. 优雅退出为什么要先停止接收新任务？
4. 健康检查为什么不能只返回进程还活着？

> 速记层
> 协议层处理 HTTP/MCP，应用服务编排用例，领域层维护业务规则，仓库封装持久化。后台队列通常提供至少一次等传递语义，消费者必须幂等并处理毒消息。可靠性来自状态、超时、重试、观测和恢复闭环，不来自“用了某个组件”。

1. 依赖方向

```
class RiskService:
    def __init__(self, repository, scorer):
        self.repository = repository
        self.scorer = scorer

    def analyze(self, query):
        rows = self.repository.fetch(query)
        return self.scorer.calculate(rows)
```

Flask 视图和 MCP Tool 把输入转为 query，调用同一服务，再转换输出。服务不导入 Flask request，也不拼飞书卡片。这样可以直接单测并复用。

层次不是越多越好。小型项目可以合并，但应保持职责和依赖方向，避免一个函数同时解析 HTTP、执行 SQL、算指标和发消息。

2. 后台任务语义

典型流程：生产者提交任务，Broker 持久化并投递，Worker 获取、执行、确认。Worker 在完成业务后、确认前崩溃，任务可能再次投递，因此“至少一次”消费者必须能重复执行。

任务记录应包含：

- 稳定任务 ID 和幂等键。
- 状态、版本和尝试次数。
- 创建、领取、开始、完成时间。
- 错误类型和可重试判断。
- 输入摘要与结果引用。

不要把巨大二进制、数据库密码或不可序列化对象直接塞入消息。

3. 领取、租约和心跳

Worker 领取任务后可能失联。租约给任务一个有限拥有时间，Worker 定期续租；租约过期后其他 Worker 可接管。必须防止旧 Worker 恢复后继续写结果，可用版本 / fencing token 条件更新。

> 深挖层：Exactly-once 是端到端属性
> 消息系统宣称的 exactly-once 往往只覆盖特定日志或事务边界。数据库写入、HTTP 调用和通知组合后仍可能重复。工程上通常通过至少一次投递、幂等消费、唯一约束和可对账状态达到等效业务结果。

4. 优雅退出

接到终止信号后常见顺序：

1. 标记实例不再就绪，停止接收新请求或任务。
2. 等待正在处理的工作到截止时间。
3. 取消可取消任务，回滚或释放租约。
4. 刷新必要日志和指标。
5. 关闭连接池和进程。

退出无限等待会阻塞发布；立即杀死则增加重复和中间状态。需要明确最大宽限期和可恢复设计。

5. 健康与可观测性

- Liveness：进程是否需要重启。
- Readiness：是否能接受新流量。
- Startup：慢启动阶段是否还在初始化。

Readiness 可以检查关键初始化和容量，但不要每次执行昂贵全链路查询。数据库偶发抖动时立即让全部实例退出负载也可能放大故障。

> 项目映射
> 自动化平台的任务调度应把"排队、领取、运行、结果上传、报告生成"拆成可观察状态。Playwright 进程崩溃后，平台根据租约重新调度；结果写入用任务版本防止旧 Worker 覆盖新结果。

6. 面试与练习

问：为什么 Service 层不能返回 Flask Response？

这会让业务用例依赖 HTTP 框架，难以被 MCP、CLI 和测试复用。Service 返回领域结果或抛领域异常，由适配层决定协议表示。

> 练习
>
> 1. 把一个视图函数拆为适配、服务和仓库，并为服务写单元测试。
> 2. 设计 Worker 在完成写入后、确认消息前崩溃的恢复测试。
> 3. 为任务租约设计 fencing token 条件更新。

官方参考：Flask background tasks with Celery；Celery tasks

第三篇：pytest 与自动化测试

C01 测试分层与 TDD

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. 单元测试的“单元”一定是一个函数吗？
2. 测试金字塔是固定数量比例吗？
3. TDD 的红、绿、重构各自要验证什么？
4. 高覆盖率为什么仍可能漏掉核心风险？

> 速记层
> 测试分层按反馈速度、隔离范围和风险选择，不按文件名分类。单元测试验证可控边界内的行为，集成测试验证组件协作，E2E 验证关键用户链路。TDD 先用失败测试明确行为，再写最小实现，最后在测试保护下重构。

1. 测试层级

层级	主要验证	常见代价
单元	业务规则、边界、错误语义	可能过度 Mock，遗漏集成问题
组件 / 集成	数据库、HTTP、消息、框架协作	环境和数据管理成本
E2E	关键用户旅程和部署配置	慢、脆弱、定位信息少

“单元”是一个可隔离行为边界，可以是函数、类或一组协作对象。测试金字塔表达底层测试数量多、反馈快的方向，不规定精确比例。数据管道、SDK、UI 产品会有不同形状。

2. TDD 循环

以查询窗口为例，先写失败测试：

```
import pytest


def test_query_window_rejects_zero_days():
    with pytest.raises(ValueError, match="days"):
        QueryWindow(days=0)
```

确认测试因为目标行为缺失而失败，而不是导入错误或测试写错。然后写最小实现让它通过，再重构重复校验。每轮保持变化小，失败原因单一。

3. 行为测试而非实现复刻

脆弱测试常断言内部私有调用顺序，而用户可观察行为并不要求：

```
def test_report_contains_ranked_users(service):
    result = service.analyze(query)
    assert [row.user_id for row in result.top_users] == [7, 3]
```

如果缓存调用次数是性能或幂等契约，可以断言；若只是当前实现细节，重构不应导致测试失败。

4. 风险驱动

优先覆盖：

- 金额、权限、状态机和数据口径。
- 空值、边界值、重复请求和并发。
- 外部依赖失败、超时和恢复。
- 曾经发生的生产缺陷。
- 变更频繁且影响范围大的模块。

覆盖率只能提示哪些代码没执行，不能证明断言有效，也不能覆盖缺失需求。变异测试可以通过故意改变实现检验测试是否真的能发现错误，但成本较高。

> 深挖层：可测试性是设计反馈
> 如果一个单元测试必须启动网络、读取全局配置并连接生产数据库，通常说明依赖和副作用没有明确边界。TDD 的价值不仅是多写测试，还会推动小接口、依赖注入、纯计算与 I/O 分离。

5. 项目应用

提审服指标计算可用纯函数单测空样本、并列、极端分布和精度；仓库集成测试验证 SQL；Flask / MCP 适配测试验证 Schema；只保留少量端到端测试验证飞书触发到结果卡片的关键链路。

6. 面试与练习

问：什么时候不适合先写单元测试？

探索未知 API 或界面时可以先做短期 spike，但进入可交付实现前仍需把得到的行为转成测试。纯文档、一次性迁移和视觉审美也需要不同验证方式，不必机械套 TDD。

> 练习
>
> 1. 为 HHI 计算按 TDD 写空输入、均匀分布和单点集中测试。
> 2. 审查一个只断言 Mock 调用的测试，补上可观察结果。
> 3. 把当前项目测试按单元、集成、E2E 和未知风险分类。

官方参考：pytest good practices

C02 pytest 用例组织与断言

速记复习 10-15 分钟；完整阅读约 18 分钟；深挖约 25 分钟

先闭卷回答

1. pytest 收集用例时会执行哪些模块代码？
2. `xfail` 和 `skip` 有什么语义区别？
3. 为什么普通 `assert` 能显示丰富差异？
4. marker 未注册会带来什么治理问题？

> 速记层
> pytest 先发现并导入测试模块，再收集测试项、准备 fixture、执行和报告。原生 assert 会被重写以提供表达式差异。skip 表示当前不执行，xfail 表示已知预期失败；marker 应注册并形成可审查的测试分类。

1. 收集和导入副作用

默认命名规则会寻找测试文件、类和函数。收集阶段需要导入模块，因此测试文件顶层不能连接外部服务或读取必需生产环境变量。可以先检查：

```
pytest --collect-only -q
```

若收集都失败，问题通常是导入路径、插件、语法或配置，而不是测试逻辑。

2. 断言

```
def test_ranked_users():
    actual = [7, 3, 9]
    assert actual == [7, 3, 9]
```

pytest 对 assert 表达式进行重写，失败时显示左右值和集合差异。避免只写 `assert result`，除非真假就是完整契约；最好给出业务可解释的精确断言。

异常断言：

```
def test_invalid_days():
    with pytest.raises(ValueError, match="between 1 and 90"):
        QueryWindow(days=100)
```

既验证异常类型，也验证稳定消息片段或属性。不要把整个易变文案写死。

3. 参数化

```
@pytest.mark.parametrize(
    ("days", "valid"),
    [(0, False), (1, True), (90, True), (91, False)],
    ids=["zero", "lower", "upper", "too-large"],
)
def test_query_window_boundaries(days, valid):
    ...
```

参数化适合相同行为的输入矩阵。若每行需要完全不同准备和断言，拆成独立测试更清楚。

4. skip、xfail 和严格性

- skip：环境或条件不满足，测试没有运行。
- xfail：已知缺陷或尚未支持，失败是预期的。
- XPASS：标记预期失败的测试意外通过，可能说明缺陷已修复，也可能测试失效。

使用 strict xfail 可以让 XPASS 触发失败，防止永久遗留无效标记。每个 xfail 应关联原因、Issue 和清理条件。

5. Marker 与配置

```
[tool.pytest.ini_options]
addopts = "-ra --strict-markers"
markers = [
  "integration: requires real infrastructure",
  "e2e: browser end-to-end tests",
]
```

可通过 `pytest -m 'not e2e'` 选择。Marker 不自动隔离环境，也不证明测试属于某层；它是运行和治理标签。

> 深挖层：Hook 和插件系统
> pytest 通过插件和 Hook 扩展收集、fixture、报告及命令行。`conftest.py` 是目录作用域插件。自定义 Hook 应使用公开规范，避免依赖私有 `_pytest` 实现；插件顺序冲突需用最小环境复现。

6. 项目与面试

自动化平台存储 pytest Node ID 时要注意参数化 ID、文件移动和插件都会影响标识。平台应保留仓库 commit、环境和收集清单，不能只存函数名。

问：pytest 为什么不需要继承 TestCase？

它通过发现规则、普通 assert、fixture 和插件组织测试，不要求测试类继承框架基类；仍可兼容部分 unittest 用法，但 fixture 和参数化能力存在边界。

> 练习
>
> 1. 为四个边界值设计可读参数 ID。
> 2. 注册 integration marker，并让未知 marker 在 CI 中失败。
> 3. 故意制造 XPASS，观察 strict 与非 strict 行为。

官方参考：pytest usage；Assertions；Markers

C03 fixture、作用域与测试数据

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. fixture 的作用域是缓存对象生命周期，还是代码可见范围？
2. yield fixture 的清理在测试失败时是否执行？
3. session fixture 为什么容易造成状态泄漏？
4. xdist 下 session fixture 是否全局只创建一次？

> 速记层
> fixture 建立显式依赖并管理准备 / 清理。scope 决定同一测试进程中的缓存生命周期；作用域越大，速度可能越快，状态污染风险越高。xdist 的每个 Worker 是独立进程，各自拥有 session fixture。

1. 依赖图

```
@pytest.fixture
def repository():
    return InMemoryRepository()


@pytest.fixture
def service(repository):
    return RiskService(repository)
```

pytest 根据参数名解析 fixture 依赖，按图创建并缓存。fixture 应返回测试真正需要的抽象，避免一个“万能环境 fixture”同时启动数据库、浏览器和消息服务。

2. yield 清理

```
@pytest.fixture
def temp_user(api_client):
    user = api_client.create_user()
    yield user
    api_client.delete_user(user.id)
```

yield 后清理通常在测试结束时执行，包括断言失败。准备过程中如果在 yield 前半途失败，尚未注册的清理步骤可能不执行。多个资源可用 `contextlib.ExitStack` 边创建边登记清理。

清理失败也应报告，但不能覆盖测试原失败。资源删除需幂等，以应对部分创建和重复清理。

3. 作用域

function、class、module、package、session 控制缓存多久。大作用域对象若可变，测试顺序会影响结果：

```
@pytest.fixture(scope="session")
def shared_config():
    return {}
```

除非每个测试只读或主动复制，否则不应共享。数据库服务进程可以 session 级启动，但每个测试的数据事务或命名空间仍应 function 级隔离。

4. 工厂 fixture

```
@pytest.fixture
def user_factory(repository):
    created = []

    def create(**overrides):
        user = repository.create_user(**overrides)
        created.append(user)
        return user

    yield create

    for user in reversed(created):
        repository.delete_user(user.id)
```

工厂允许测试只描述差异，并集中清理。默认数据要显式稳定，不要依赖当前时间或随机值而没有记录 seed。

5. 临时目录和端口

使用 `tmp_path` / `tmp_path_factory` 获得隔离目录，不要在仓库固定目录写 `output.json`。动态端口存在“找到空闲端口后被其他进程抢占”的竞态，最好让服务绑定端口 0 并读取实际端口。

> 深挖层：fixture 缓存键
> fixture 缓存与测试节点、作用域、参数化和 Worker 有关。把 session 理解成“整个 CI 只有一次”是错误的：多进程、分片和重跑会创建多份。需要真正全局唯一资源时必须使用外部协调，并设计崩溃清理。

> 项目映射
> Playwright 每个测试使用独立 BrowserContext，浏览器进程可按 Worker 复用；登录状态、下载目录和 Trace 仍要按用例隔离。数据库测试使用 Worker 前缀和事务清理，避免五个 xdist Worker 抢同一记录。

6. 面试与练习

问：fixture scope 越大越快，为什么不用 session？

共享越多，隔离越弱，顺序依赖和并发污染越难定位。应复用昂贵且可安全共享的基础设施，隔离每个测试可变数据。

> 练习
>
> 1. 把一个 session 级可变字典改为只读模板加 function 复制。
> 2. 模拟 fixture 准备到一半失败，使用 ExitStack 保证已创建资源清理。
> 3. 运行 xdist，记录每个 Worker 的 session fixture 实例。

官方参考：pytest fixtures；Temporary paths

C04 参数化、Mock、Patch 与 Monkeypatch

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. Patch 应该替换对象定义处还是被测代码查找处？
2. Mock 的 spec / autospec 解决什么问题？
3. Monkeypatch 如何保证测试后恢复？
4. 过度 Mock 为什么会出现“测试全过、集成全坏”？

> 速记层
> Mock 用可控替身隔离慢、随机或失败依赖；Patch 必须替换被测模块实际查找的名称。spec / autospec 限制接口漂移。Monkeypatch 临时修改属性、字典、环境和路径并在测试后恢复。优先注入依赖，少 Patch 全局。

1. 在查找处 Patch

```
# service.py
from gateway import send_message


def notify(payload):
    return send_message(payload)
```

测试应 Patch `service.send_message`，因为函数运行时从 service 模块全局名称查找，而不是再访问 `gateway.send_message`。

更清晰的设计是注入 gateway：

```
def notify(payload, gateway):
    return gateway.send_message(payload)
```

这样测试传入小型 Fake，无需知道模块导入细节。

2. spec 与 autospec

没有 spec 的 Mock 接受任意属性和参数，真实接口删改后测试可能继续通过。autospec 根据目标签名约束调用，但仍不能验证真实网络协议、序列化和副作用。

```
from unittest.mock import create_autospec

gateway = create_autospec(MessageGateway, instance=True)
gateway.send.return_value = "msg-1"
```

3. Monkeypatch

```
def test_uses_test_endpoint(monkeypatch):
    monkeypatch.setenv("API_URL", "https://example.invalid")
    monkeypatch.setattr(service, "clock", FakeClock())
```

fixture 会在测试后撤销修改。修改标准库或 pytest 自身函数可能破坏框架，可使用 `monkeypatch.context()`

缩小范围。

4. Fake、Stub、Mock 的取舍

- Stub：返回预设值。
- Fake：有简化但可工作的实现，例如内存仓库。
- Mock：记录并验证交互。
- Spy：包装真实对象并观察调用。

术语在团队中可能略有不同，关键是替身是否模拟状态、是否验证交互、与真实接口偏差多大。复杂仓库更适合 Fake 加契约测试，而不是几十个链式 Mock。

5. 时间与随机性

不要在每个测试到处 Patch `datetime.now`。可以注入 Clock：

```
class Clock(Protocol):
    def now(self) -> datetime: ...
```

随机算法接受显式 seed 或 Random 实例。测试不应依赖真实睡眠，重试器可以注入 sleeper 并断言退避序列。

> 深挖层：交互测试的脆弱性
> 断言每个私有调用和精确顺序会把实现复制进测试。只有当调用本身是契约，例如“支付网关只扣款一次”“事务失败不得发送通知”，才值得严格验证。其余优先断言输出和状态。

> 边界案例：异步 Mock
> 异步函数需要可 await 的替身并验证 `await`，上下文管理器和异步迭代器还要配置对应魔术方法。普通 Mock 返回值可能让测试在错误层失败或根本未执行异步行为。

6. 项目与面试

MCP Tool 单测可以注入内存 Repository 和 Fake Context，验证结构化结果；另有一组协议集成测试启动真实 Server / Client。只 Mock FastMCP 所有装饰器无法证明 Schema 注册正确。

问：什么时候不要 Mock？

纯函数无需 Mock；稳定快速的内存对象可直接用；数据库驱动、序列化和框架生命周期应保留适量真实集成测试，防止替身与真实行为漂移。

> 练习
>
> 1. 复现 Patch 错模块导致替换无效，再改为查找处 Patch。
> 2. 用 autospec 让错误参数调用在测试中失败。
> 3. 把 Mock 仓库重构为内存 Fake，并增加与真实仓库的契约测试。

官方参考：Where to patch；pytest monkeypatch

C05 API、数据库与集成测试

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. Flask 测试客户端是否真的通过 TCP 请求服务？
2. 数据库测试用事务回滚时，有哪些场景不能被覆盖？
3. 契约测试和端到端测试分别验证什么？
4. 测试使用 SQLite 替代 MySQL 会遗漏哪些风险？

> 速记层
> 集成测试验证真实组件边界：路由、序列化、数据库、消息和配置。Flask Test Client 在进程内调用 WSGI 应用，速度快但不覆盖真实网络和 WSGI Server。数据库测试应尽量使用生产同类引擎，并为数据、事务和并发提供隔离。

1. Flask 测试客户端

```
def test_create_query(client):
    response = client.post(
        "/api/queries",
        json={"package_id": "pkg-1", "days": 7},
    )
    assert response.status_code == 201
    assert response.json["package_id"] == "pkg-1"
```

它验证路由、上下文、校验、错误处理和响应转换，但不验证反向代理 Header、TLS、实际端口、Worker 超时和部署配置。少量部署级冒烟测试补足这些边界。

2. 数据库隔离

常见策略：

- 每测试事务，结束回滚：快，但被测代码自己提交、开启新连接时可能逃逸。
- 每测试清表：接近真实提交，成本高且并发易冲突。
- 每 Worker 独立 Schema / Database：适合并行，创建与迁移有成本。
- 容器化临时数据库：隔离强，但启动和资源管理更复杂。

选择要覆盖真实事务行为。用 SQLite 测 MySQL 代码会遗漏类型、锁、隔离、函数、SQL 方言和索引计划差异；SQLite 可用于纯仓库接口的快速反馈，但不能替代 MySQL 集成测试。

3. 迁移和种子数据

测试数据库应通过正式迁移创建，而不是手写另一份 Schema。种子数据要最小、可读，并在测试中明确创建与业务相关部分。共享巨型 SQL Dump 会隐藏依赖并拖慢反馈。

4. 外部 API

分层验证：

- 单元测试使用 Fake / Stub。
- 契约测试验证请求和响应 Schema、认证及错误映射。
- 沙箱集成测试验证真实服务，但不能成为每次本地测试必需条件。
- 生产冒烟只执行安全、只读或可清理动作。

录制回放能提高稳定性，但录制内容可能含敏感数据，也会随 API 变化过时。

> 深挖层：事务测试假象
> 外层测试事务回滚可能让代码始终看不到真实提交后的行为，例如唯一约束延迟、另一个连接可见性、提交 Hook 和 outbox Worker。关键事务流程需要真实 commit 的集成测试，并在独立数据库或命名空间清理。

5. 可重复环境

测试报告应记录应用 commit、迁移版本、Python 和依赖版本、数据库版本、环境配置摘要。仅记录“集成测试通过”无法复现。

> 项目映射
> 风险查询仓库测试可以在 MySQL 临时 Schema 中准备多表数据，验证 Join、NULL、TopN 和时间边界；服务层另用 Fake 仓库快速覆盖指标。两者职责不同，不能只保留其中一种。

6. 面试与练习

问：集成测试为什么慢？怎么优化？

慢通常来自环境启动、数据库重置、网络和过宽场景。可复用只读基础设施、缩小种子数据、并行隔离、按变更选择测试，但不能用 Mock 删掉正要验证的边界。

> 练习
>
> 1. 为 Flask API 写成功、校验失败和领域冲突测试。
> 2. 设计一个必须真实 commit 才能验证的 outbox 集成测试。
> 3. 列出 MySQL 与 SQLite 在项目查询中的不可替代差异。

官方参考：Testing Flask applications

C06 异步测试、并发执行与覆盖率

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. 异步测试结束时遗留 Task 为什么危险？
2. xdist 是线程并行还是进程并行？
3. 行覆盖率 100% 能否证明分支全部验证？
4. 并行后才失败的测试通常暴露了什么？

> 速记层
> 异步测试要管理事件循环、任务、超时和清理；测试结束时不应遗留后台 Task。pytest-xdist 使用多个 Worker 进程，收集结果必须一致，外部资源需按 Worker 隔离。覆盖率是未执行代码的线索，不是正确性证明。

1. 异步测试

```
@pytest.mark.asyncio
async def test_timeout_cancels_operation():
    with pytest.raises(TimeoutError):
        async with asyncio.timeout(0.01):
            await never_finishes()
```

具体 marker 和循环管理取决于插件。测试应使用有界超时，确保失败不会挂住套件。创建后台 Task 的组件要暴露关闭方法，并在 fixture 清理中 await。

2. xdist 资源隔离

Worker 可通过 `worker_id` 派生唯一资源：

```
@pytest.fixture(scope="session")
def schema_name(worker_id):
    return f"test_{worker_id}"
```

还需隔离账号、端口、下载目录、队列、缓存前缀和浏览器状态。并行测试不能依赖执行顺序，也不能假设每个文件在同一 Worker。

3. 收集一致性

xdist 要求 Worker 收集到兼容测试集合。按随机数据动态生成参数、依赖本地文件顺序或环境差异，会导致收集不一致。参数 ID 和数据排序应确定。

4. Flaky 的根因

并行常暴露原有缺陷：

- 共享全局状态。
- 测试未清理数据。
- 依赖顺序。

- 固定端口和文件名。
- 时间窗口太紧。
- 等待条件使用 sleep 而不是状态。

重跑可以收集证据或缓解外部偶发故障，但不能把 Flaky 标成“重跑通过即通过”。应记录首次失败和重跑结果，并设治理门槛。

5. 覆盖率

行覆盖、分支覆盖和条件覆盖回答的问题不同。下面即使两行都执行，也可能遗漏 `is_admin=False` 分支：

```
if user.is_admin:
    grant_all()
else:
    grant_limited()
```

覆盖率目标应结合风险和变更，不要为追数字写没有断言的测试。可以对新代码设门槛并审查关键未覆盖分支。

> 深挖层：并发调度不是性能证明
> `-n auto` 会增加进程、数据库连接和浏览器数量，可能让套件更慢或压垮测试环境。应测量单 Worker 时长、并行效率、共享瓶颈和失败率，选择稳定 Worker 数。Playwright 官方示例建议也只是起点，不替代项目压测。

> 项目映射
> 自动化平台可将 Worker 数作为每个执行环境的容量配置，结合浏览器内存和账号池，而不是让用户任意填写。调度器记录各 Worker 任务、资源和首次失败证据，便于定位 Flaky。

6. 面试与练习

问：并发跑测试失败，串行通过，怎么办？

先保留失败证据，检查共享状态、数据和端口，再用固定 Worker / 用例组合最小复现。不要先加 sleep 或无限重试。

> 练习
>
> 1. 制造共享文件名冲突，让 xdist 失败，再按 Worker 隔离。
> 2. 测试组件关闭后 `asyncio.all_tasks()` 不含遗留业务任务。
> 3. 为权限判断补齐分支覆盖，同时检查断言是否有意义。

官方参考：Flaky tests；pytest-xdist；coverage.py

C07 Playwright Python UI 自动化

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. Browser、Context 和 Page 分别是什么隔离层？
2. Playwright 自动等待会等待哪些条件，又不会等待什么？
3. 为什么 `time.sleep()` 容易产生 Flaky？
4. Trace、截图和录像各能回答什么问题？

> 速记层
> Browser 是浏览器进程连接，BrowserContext 提供轻量隔离会话，Page 是页面。Locator 在动作前执行可操作性检查并自动重试定位；业务数据最终一致、后台任务完成等条件仍需显式断言。失败证据应组合 Trace、页面截图、控制台和网络信息。

1. 隔离模型

```
def test_login(page):
    page.goto("https://example.test/login")
    page.get_by_label("用户名").fill("tester")
    page.get_by_role("button", name="登录").click()
    expect(page).to_have_url(re.compile(r"/dashboard$"))
```

官方 pytest 插件通常为测试提供隔离 Context / Page。可在 Worker 内复用 Browser 进程，减少启动成本；不要在测试间复用有状态 Context，除非明确管理存储和清理。

2. Locator 优先级

优先用户可感知语义：role、label、text、test id；避免长 CSS / XPath 依赖 DOM 层级。测试 ID 是开发和测试共同契约，但不能替代可访问性语义。

Locator 是延迟查询，可在每次动作 / 断言时重新解析；ElementHandle 容易引用已经替换的旧节点，普通用例优先 Locator。

3. 自动等待的边界

点击前会检查目标是否唯一、可见、稳定、可接收事件和启用等相应条件。它不会理解“报表后台计算完成”或“数据库最终一致”。应断言用户可观察状态：

```
expect(page.get_by_test_id("report-status")).to_have_text("已完成")
```

固定 sleep 要么过短失败，要么过长浪费。确需时间推进时，使用可控时钟、事件或状态轮询，并设置有意义超时。

4. 网络和状态准备

可以通过 API 创建前置数据，再在 UI 验证；这样减少慢 UI 操作，同时保留关键用户路径。路由拦截可模拟边界响应，但至少保留一组真实后端集成测试。

登录状态文件包含敏感 Cookie / Token，不应提交仓库。多 Worker 使用独立账号或可并发会话，不能共享会互踢的单账号。

5. Trace 与证据

Trace 可回看动作、DOM 快照、网络和控制台；截图展示单一时刻；录像展示时间过程但搜索能力弱。失败保留策略要控制磁盘，上传制品时脱敏。

> 深挖层：动作成功不等于业务正确
> Locator.click 成功只说明动作条件满足并执行，不证明请求成功、数据写入或通知到达。每个动作后应断言最终业务状态，而不是只验证元素存在。纯 UI 回归应围绕用户可观察结果构建。

> 边界案例：虚拟列表和滚动
> 虚拟表格只渲染视口附近行，DOM 中没有全部数据。直接统计 Locator 数量会漏行。应按产品交互滚动、使用分页 API 或验证当前视窗与总数契约，不能无限滚动后声称穷举。

6. 项目与面试

多维筛选页面需要：设置筛选、等待网络或状态、验证表格行满足条件、清空恢复默认。每个筛选器使用独立断言，同时保留组合筛选覆盖；失败附 Trace、截图、选项值和响应摘要。

问：Playwright 自动等待是否能消除所有 Flaky？

不能。它解决元素可操作性和重试断言的一部分，数据隔离、异步业务状态、第三方依赖和资源竞争仍需设计。

> 练习
>
> 1. 把一个 `sleep(3)` 用例改为针对业务完成状态的 expect。
> 2. 为失败用例保留 Trace、截图、控制台错误和请求 ID。
> 3. 设计虚拟列表滚动验证，说明覆盖边界。

官方参考：Auto-waiting；Pytest plugin；Trace Viewer

C08 自动化测试平台设计

速记复习 10-15 分钟；完整阅读约 20 分钟；深挖约 30 分钟

先闭卷回答

1. 测试平台与脚本仓库的本质差别是什么？
2. 调度器如何防止同一任务被多个 Worker 同时执行？
3. 测试结果为什么必须绑定代码和环境版本？
4. 平台重试失败用例时，如何避免掩盖首次失败？

> 速记层
> 测试平台把用例、环境、任务、调度、执行、证据和质量结果形成可追踪系统。控制面负责建模和调

> 度，执行面在隔离 Worker 运行代码。每次执行必须绑定 commit、依赖、配置和制品；状态机、租约和幂等保证失败可恢复。

1. 核心模型

建议最小实体：

- Project / Repository：代码来源和权限。
- TestPlan：选择范围、参数和环境。
- Run：一次不可变执行请求。
- Task / Shard：可调度工作单元。
- Worker：带能力和租约的执行者。
- CaseResult：用例结果、耗时和错误分类。
- Artifact：Allure、Trace、截图、日志、录像。

Run 创建后保存代码 commit 和配置快照，不能让后续编辑悄悄改变历史结果。

2. 状态机

```
queued -> leased -> running -> uploading -> finished
                      |             |
                      +-> failed <--+
```

每次转换要有条件和版本。调度器用租约防止永久占用；Worker 续租并携带 fencing token 写结果。取消是请求状态，执行器仍需协作停止和清理。

3. 分片和调度

可按文件、历史耗时或 marker 分片。只平均用例数量可能让一个慢文件拖尾；使用历史时长要处理新用例和数据漂移。存在共享 fixture 或顺序约束的用例不能随意拆分。

Worker 能力包括 Python、浏览器、设备、网络和标签。调度器匹配能力并控制全局数据库 / 账号并发，不只看 CPU 空闲。

4. 结果与证据

统一结果需要区分：passed、assertion failed、setup error、infrastructure error、cancelled、timeout、skipped、xfail。平台自身错误不能算产品缺陷，断言失败也不能全部自动重试成通过。

制品采用稳定路径和保留策略，敏感 Header、Cookie、数据库结果需要脱敏。Allure 是展示和证据之一，不是平台领域模型本身。

> 深挖层：结果最终一致
> Worker 可能先上传结果后状态更新失败，也可能状态 finished 但制品尚未可见。平台需要可重试上传、内容哈希、完成清单和对账任务。用单个布尔 success 无法表达这些中间状态。

5. Flaky 治理

记录首次结果和每次重跑，计算按用例、环境和失败签名的波动。隔离 Flaky 用例要有负责人、原因和期限；不能永久排除而不影响质量门禁。自动聚类只提供候选，不替代根因分析。

6. MVP 边界

第一阶段可聚焦：提交 pytest 任务、单环境 Worker、实时状态、Allure 与失败制品、取消和重试。复杂资源编排、智能选例、多租户计费和多模态分析应在基础状态机稳定后演进。

> 项目映射
> 你的平台路线可以先支持 Flask + pytest + Allure 接口自动化，再加入 Playwright。接口和 UI 共用 Run / Task / Result 模型，执行器插件不同；不要为 UI 另造无法汇总的结果体系。

7. 面试与练习

问：如何证明平台不是“套了页面的 pytest”？

说明它解决了多人任务建模、环境隔离、调度恢复、权限、证据留存、历史趋势和质量门禁；同时诚实说明 MVP 已完成到哪一层。

> 练习
> 1. 为 Run 和 Task 画状态机，列出每条非法转换。
> 2. 设计 Worker 崩溃、租约过期、旧 Worker 回写的测试。
> 3. 定义最小结果 Schema，并绑定 commit、环境和制品哈希。

官方参考：pytest；Playwright Python；Allure Report

第四篇：MCP 与 AI 应用

D01 MCP 工作模型、协议与版本边界

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. Host、Client、Server 各自负责什么？
2. Tool、Resource、Prompt 的控制权分别偏向谁？
3. stdio 为什么绝不能把普通日志写到 stdout？
4. 2026-07-28 版协议为何不再依赖握手与会话？

> 速记层
> MCP 是让 AI Host 用统一协议发现和调用外部能力的边界，不是 Agent 框架。Host 管用户、模型与权限；Client 代表 Host 对接一个 Server；Server 暴露 Tool、Resource、Prompt。协议基于 JSON-RPC 2.0。当前 2026-07-28 规范以无会话、自包含请求和逐请求能力信息为主；旧客户端仍可能走 initialize 会话，服务端要明确兼容边界。

1. 三个角色

角色	主要责任	不应偷偷承担
Host	模型调用、用户授权、上下文编排、界面	替 Server 猜业务权限
Client	协议适配、发现、调用、版本与传输处理	绕过 Host 的同意策略
Server	能力实现、输入校验、业务授权、审计	相信模型已经验证参数

一个 Host 内可以有多个 Client，每个 Client 对接一个逻辑 Server。Server 应聚焦清晰领域，例如“风险数据查询”，而不是把整个内网变成一个万能工具。

2. 三种服务端原语

原语	含义	典型控制者	例子
Tool	可执行操作，可能有副作用	模型选择，Host 可审批	生成风险报告、创建任务
Resource	通过 URI 读取上下文	应用或用户选择	指标定义、报告模板、数据字典
Prompt	参数化消息或工作流模板	用户显式选择为主	“审查这份报告”模板

不要只因“都能返回文本”就全部做成 Tool。读取稳定文档适合 Resource；会更改状态、耗费资源或做计算的动作适合 Tool；可复用交互入口适合 Prompt。

3. JSON-RPC 消息

请求有 `id`，响应必须关联同一 `id`；通知没有 `id`，不期待响应。协议错误和业务失败要区分：无效参数可用协议错误；合法请求得到“数据不存在”通常应返回结构化业务结果。

```
{
  "jsonrpc": "2.0",
  "id": 17,
  "method": "tools/call",
  "params": {
    "name": "risk_summary",
    "arguments": {"project_id": "P-42"}
  }
}
```

2026-07-28 版普通结果带 `resultType: "complete"`；需要用户或客户端补充输入时可返回 `resultType: "input_required"`，由 Client 带回答重试原请求。这个 Multi Round-Trip Request 模式替代了新协议中的服务端反向请求。

4. 传输不是业务语义

- stdio：Host 启动本地子进程，通过 stdin/stdout 传协议帧。stdout 只能放协议数据，日志写 stderr。
- Streamable HTTP：适合远程部署、认证、网关和水平扩展。超时、断流和重试仍要由业务幂等性兜底。
- SSE：官方 Python SDK 仍可用于旧兼容路径，但新系统应优先核对当前规范与 SDK 推荐传输。

传输层成功不表示 Tool 业务成功；HTTP 200 里仍可能有工具错误。反过来，网络超时也不证明服务端没有完成副作用。

5. 当前版与旧版的关键差异

> 版本说明：截至 2026-08-30
> MCP 2026-07-28 请求自带协议版本与能力元数据，使用 `server/discover` 做发现，不要求 2025 时代的 initialize 握手和 `Mcp-Session-Id`。官方 Python SDK 2.x 可同时服务新旧版本；旧客户端仍可能需要会话和负载均衡粘性。不要把某个桌面客户端当前行为误写成协议永久规则。

新协议移除 `ping`，并把 Tasks 移到可选扩展。Roots、Sampling 和 MCP 协议日志已经被标记为弃用；维护旧系统时仍可能遇到它们。学习顺序应是：先掌握核心原语和安全边界，再按目标客户端确认版本能力。

> 深挖层：无会话不等于无状态业务
> 协议请求可以无会话，但你的业务仍可能有任务、幂等键和权限状态。多轮请求的密封 request_state、长任务句柄和数据库事务分别属于协议续接、业务执行和持久化三层，不应混为一个“session”。

> 边界案例：重试一个已扣费工具
> Client 因响应断流重发请求，新请求 ID 不会自动防止重复扣费。工具必须接受业务幂等键，并持久化请求指纹与最终结果；检测到同键不同参数时应拒绝。

6. 项目与面试

风险分析项目可把“指标定义”做 Resource，“执行聚合查询”做只读 Tool，“生成审查提纲”做 Prompt。Host 决

定是否让模型调用，Server 仍独立检查调用者是否能读取该项目。

问：MCP 和普通 REST API 有何不同？

REST 面向通用服务集成；MCP 在 JSON-RPC 之上约定了 AI Host 可发现的 Tool、Resource、Prompt、能力协商和交互模式。底层业务服务仍可保留 REST，MCP Server 作为受控适配层，而非复制全部业务。

> 练习
>
> 1. 把一个“查数据并发邮件”的万能 Tool 拆成读、预览和确认执行三步。
> 2. 分别画出 stdio 与 Streamable HTTP 的故障边界。
> 3. 解释新旧协议同时在线时，为什么不能只在开发机测试。

官方参考：MCP 2026-07-28 Specification；Architecture；Key Changes

D02 用官方 Python SDK 与 FastMCP 构建服务

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 类型标注如何变成 Tool 的输入 Schema？
2. 为什么协议装饰器里不应直接堆 SQL 和权限逻辑？
3. 官方 `mcp` 包和独立 `fastmcp` 包是什么关系？
4. 本地进程内测试能证明哪些事，不能证明哪些事？

> 速记层
>
> 官方 Python SDK 2.x 使用 `MCPServer`，从签名、类型标注和文档生成工具契约；`Client` 统一处理内存、stdio 和 HTTP。独立 FastMCP 框架也提供 Server、Client、中间件和依赖注入。无论选哪套，装饰器层只做协议适配，业务服务保持普通 Python，可单测、可复用、可替换传输。

1. 最小官方 SDK 服务

```
from dataclasses import dataclass

from mcp.server import MCPServer

mcp = MCPServer("risk-tools")


@dataclass(frozen=True)
class RiskSummary:
    project_id: str
    score: float
    level: str


@mcp.tool()
def summarize_risk(project_id: str, threshold: float = 0.7) -> RiskSummary:
    """Return a read-only risk summary for one authorized project."""
    score = 0.82  # 示例；真实实现调用业务服务
    return RiskSummary(
        project_id=project_id,
        score=score,
        level="high" if score >= threshold else "normal",
```

```
    )


@mcp.resource("risk-policy://{version}")
def risk_policy(version: str) -> str:
    """Return the reviewed risk-policy text for a version."""
    return f"policy version: {version}"
```

开发检查可运行 `uv run mcp dev server.py`；远程服务可用 `mcp run ... --transport streamable-http`。客户端通过 `Client` 调用时读取 `structured_content`，不要从展示文本反解析字段。

```
import asyncio

from mcp import Client


async def main() -> None:
    async with Client("http://localhost:8000/mcp") as client:
        result = await client.call_tool(
            "summarize_risk",
            {"project_id": "P-42", "threshold": 0.75},
        )
        print(result.structured_content)


asyncio.run(main())
```

2. Schema 来自契约，不来自猜测

类型标注约束结构，docstring 解释语义。`float` 不能表达“0 到 1”，字符串也不能表达项目 ID 是否存在；需要使用 SDK 支持的约束类型、枚举或显式业务校验。默认值意味着可选行为，应稳定且安全。

返回值尽量是 dataclass、TypedDict 或模型对象，让结构化输出有确定字段。展示文本可以给人看，但 Agent 的下一步应消费结构化数据。

3. 保持薄适配层

```
class RiskService:
    def __init__(self, repository: "RiskRepository") -> None:
        self.repository = repository

    def summary(self, project_id: str, actor_id: str) -> RiskSummary:
        if not self.repository.can_read(actor_id, project_id):
            raise PermissionError("project is not visible")
        return self.repository.load_summary(project_id)
```

协议 Handler 负责拿到身份、调用服务、翻译错误；Repository 负责数据库。这样 Flask 路由、CLI 和 MCP Tool 可以共享同一用例层，权限规则也不会出现三份。

4. 两个相似名称

包	当前定位	典型导入
`mcp`	Model Context Protocol 官方 Python SDK 2.x	`from mcp.server import MCPServer`
`fastmcp`	独立的 MCP 应用框架	`from fastmcp import FastMCP`

旧版官方 SDK 曾内置名为 FastMCP 的高层类，2.x 已更名为 `MCPServer`。看到教程时先核对包名、主版本和

发布日期；不要把两套 API 拼接。

5. 生命周期和依赖

数据库连接池、HTTP Client、配置和指标对象应在服务生命周期创建一次，在退出时关闭；请求级事务和身份放在上下文或依赖中。模块导入时直接连数据库会破坏测试收集和 CLI 探测。

> 深挖层：装饰器做了什么
> 高层 SDK 会检查 Callable 签名，生成 JSON Schema，注册名称到 Handler 映射，并在调用时完成反序列化、校验、依赖解析和结果编码。它不理解你的业务不变量；Schema 通过仍需领域校验。

> 边界案例：新增一个可选参数
> 对 Python 调用者是向后兼容的默认参数，对缓存的工具 Schema 或严格客户端未必立即可见。列表变更通知、TTL、客户端重新发现和版本化名称都要按目标协议验证。

6. 项目与面试

把“查询风险指标”用例写成普通 Service，先有纯单元测试；再各写 Flask 路由和 MCP Tool 的契约测试。面试演示两种入口得到相同领域结果，并说明身份来源不同。

问：为什么不直接把所有函数加上 `@mcp.tool()`？

工具面是公开契约和攻击面。只暴露最小业务能力，避免泄露内部辅助函数、表结构和无权限边界的通用执行器。

> 练习
>
> 1. 为 RiskSummary 增加可解释的证据字段，并保持输出可分页。
> 2. 给 Handler、Service、Repository 分别写出一条职责边界。
> 3. 找一篇旧教程，标出 v1 和 v2 导入路径差异。

官方参考：MCP Python SDK；Python SDK Documentation；FastMCP Documentation

D03 Tool 接口、Schema 与副作用设计

速记复习 10-15 分钟；完整阅读约 14 分钟；深挖约 24 分钟

先闭卷回答

1. 一个 Tool 名称和描述怎样降低模型误调用？
2. 为什么 `execute_sql(sql: str)` 是危险的默认设计？
3. Tool 超时后，客户端能否判断副作用是否发生？
4. 结构化错误至少应包含哪些字段？

> 速记层
> Tool 是给不稳定调用者使用的稳定 API。名称用动词和领域对象；Schema 缩小选择空间；描述写前置条件、副作用、权限和返回语义；输出结构化且有界。读写工具分开，高风险操作采用预览 - 确认 - 执行，

> 写操作接受幂等键并保留审计。

1. 名称与语义

`run`、`process`、`do_task` 几乎不给模型区分信号。更好的是：

- `get_project_risk_summary`：只读、单项目、摘要。
- `preview_risk_report_export`：计算但不提交。
- `create_risk_report_export`：产生持久副作用。

描述不要写营销话术，要回答：何时调用、何时不要调用、需要什么授权、是否修改状态、结果是快照还是实时值。

2. 用 Schema 消除非法状态

```
from enum import StrEnum
from typing import TypedDict


class Window(StrEnum):
    DAY_7 = "7d"
    DAY_30 = "30d"


class RiskQuery(TypedDict):
    project_id: str
    window: Window
    include_evidence: bool
```

枚举优于“请传 7d 或 30d”的自然语言约定；两个互斥布尔值不如一个枚举。时间要标明时区和左右边界；金额要标明币种和最小单位；分页要返回 next_cursor 而不是让模型猜页码。

3. 结果和错误

```
from typing import Literal, TypedDict


class ToolError(TypedDict):
    code: Literal[
        "invalid_input",
        "forbidden",
        "not_found",
        "conflict",
        "temporarily_unavailable",
    ]
    message: str
    retryable: bool
    request_id: str
```

对人可读 message 不能替代机器字段。不要把数据库异常全文返回给模型；内部日志记录堆栈、SQL 指纹和 request_id，外部只给稳定分类和安全信息。

4. 副作用与幂等

对写操作要求 `idempotency_key`，存储 `(actor, tool, key, input_hash, status, result)`。同键同参数返回旧结果；同键不同参数返回 conflict。执行中崩溃要能继续查询状态，而不是盲目再次执行。

高风险操作可拆为：

1. preview 返回待执行内容、影响范围和短期确认令牌；
2. Host 展示并获得用户同意；
3. execute 校验令牌、输入摘要、身份、有效期和幂等键；
4. 返回审计 ID 和最终状态。

确认令牌不能只包含可修改的明文 JSON，应由服务端保存或签名，并绑定调用者与精确参数。

5. 输出预算

数据库几万行、完整日志或 Base64 文件会挤占模型上下文。Tool 返回总数、摘要、有限条目、截断标志和游标；大制品存对象存储，返回受权限控制的 Resource URI 或短期引用。

> 深挖层：Schema 是选择提示也是安全边界的一部分
> 模型通常根据名称、描述和 Schema 选择工具，但它可能传恶意或矛盾参数。Schema 只能挡结构错误，服务端仍需鉴权、领域不变量和资源限制。把“模型大概率不会这样调用”当控制措施是设计缺陷。

> 边界案例：返回 200 但业务排队
> 长任务创建成功不等于报告已生成。返回 `job_id`、`state="queued"` 和状态查询方式；不要把“已提交”写成“已完成”。若使用 Tasks 扩展，仍需确认 Host 和 Server 都声明支持。

6. 项目与面试

风险工具不暴露任意 SQL，而暴露领域查询参数：项目、时间窗、指标、维度、上限。白名单编译到参数化 SQL，结果附口径版本和查询时间。

问：如何设计一个删除资源 Tool？

先问能否做软删除或专用业务动作；若必须删除，提供精确资源 ID、预览影响、用户确认、幂等键、权限二次检查、审计和可恢复窗口。不能接受模糊名称批量匹配后直接执行。

> 练习
> 1. 重写 `process(data: dict) -> str` 的名称、输入与输出。
> 2. 设计超时后可查询的写工具状态模型。
> 3. 给批量导出加入最大行数、游标和截断语义。

官方参考：MCP Tools；MCP Security Best Practices

D04 SQL 与数据分析能力的安全封装

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 参数化 SQL 为什么仍不能安全处理任意表名？
2. 只读数据库账号为什么仍可能造成事故？
3. 指标口径、SQL 和展示字段如何保持可追踪？
4. 如何阻止一次合法查询拖垮数据库？

> 速记层
> 让 Agent 直接生成并执行任意 SQL，既暴露 Schema，也放大越权和资源耗尽风险。更稳的方式是领域 DSL + 白名单编译器 + 参数绑定 + 只读账号 + 行数、时间和成本上限。每个结果附口径版本、数据时间、截断状态和审计 ID。

1. 从领域参数到 SQL

```
from dataclasses import dataclass
from datetime import date


ALLOWED_METRICS = {
    "high_risk_count": "SUM(risk_level = 'high')",
    "avg_score": "AVG(risk_score)",
}


@dataclass(frozen=True)
class MetricQuery:
    project_id: str
    metric: str
    start: date
    end: date


def compile_query(query: MetricQuery) -> tuple[str, tuple[object, ...]]:
    expression = ALLOWED_METRICS.get(query.metric)
    if expression is None:
        raise ValueError("unsupported metric")
    if query.end < query.start or (query.end - query.start).days > 366:
        raise ValueError("invalid time window")
    sql = f"""
        SELECT {expression} AS metric_value
        FROM risk_event
        WHERE project_id = %s
          AND event_date >= %s
          AND event_date < %s
    """
    return sql, (query.project_id, query.start, query.end)
```

值通过绑定参数；列名、表名和表达式不能用参数占位，必须来自服务端白名单。这里的 f-string 只插入内部固定表达式，不接收用户文本。

2. 多层限制

- 身份：从认证上下文获得 actor，不允许模型自报 `user_id`。

- 范围：先解析 actor 可见项目，再将项目条件写入查询。
- 账号：专用只读用户，只授权必要视图而非原始全库。
- 资源：连接池上限、查询超时、最大扫描窗口、最大返回行数。
- 输出：字段白名单、脱敏、游标、对象大小和下载有效期。
- 审计：记录工具、调用者、参数摘要、SQL 指纹、耗时、行数和结果分类。

只读 SQL 仍能做全表扫描、锁或消耗临时空间；只读账号不等于无害。分析流量最好走只读副本或专用分析库，并接受其延迟语义。

3. 指标口径

每个指标应有：唯一名称、自然语言定义、分子分母、过滤条件、时间字段、时区、空值策略、版本和负责人。SQL 是实现，不是口径本身。口径改变时保留版本，使历史报告可解释。

```
result = {
    "metric": "avg_score",
    "definition_version": "2026-08-v2",
    "value": 0.731,
    "as_of": "2026-08-30T09:00:00Z",
    "rows_scanned": 1240,
    "truncated": False,
}
```

4. 是否允许自然语言查询

可以让模型把自然语言映射到有限 DSL，再由确定性代码验证。若产品确需生成 SQL，应先解析 AST，限定语句类型、表、列、函数、JOIN、子查询和 LIMIT；使用只读隔离环境并先做成本评估。正则无法可靠解析 SQL 语法。

> 深挖层：行级权限必须落到执行路径
> 只在 Tool 描述中写“用户只能访问自己的项目”没有效果。权限条件必须由服务端根据可信身份注入查询，或由数据库视图 / Row Level Security 强制。若让模型提供 project_id 后只检查格式，就形成 IDOR 越权。

> 边界案例：只读副本上的“最新数据”
> 副本可能有秒级或更长延迟。结果必须附 `as_of` 或复制延迟；刚创建后立即查询的 read-after-write 流程可走主库、等待可见性或返回 pending，不能悄悄报“没有数据”。

5. 项目与面试

实现 `query_risk_metrics` 时，先支持 5 个高价值指标、2 个维度和 2 种时间窗，建立黄金结果集。比“支持任意 SQL”更容易验证正确性、权限和性能，也更能讲清产品边界。

问：参数化查询是否已经解决 SQL 注入？

它解决值拼接的主要注入风险，但动态标识符、动态排序、表达式、权限范围和资源耗尽仍需白名单与限制；错误地把 SQL 片段当值拼接仍然危险。

> 练习

> 1. 给 compile_query 增加白名单维度和固定排序。
> 2. 设计一个跨租户 project_id 的越权测试。
> 3. 列出查询超时、结果截断和副本延迟的外部语义。

官方参考：MySQL Prepared Statements；MySQL EXPLAIN；MCP Tools

D05 Agent 工作流的可靠性、评测与人工确认

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 为什么不能用一次 Demo 的成功率评价 Agent？
2. 哪些步骤交给模型，哪些步骤必须确定性执行？
3. 如何区分模型错误、工具错误和基础设施错误？
4. 人工确认应绑定哪些上下文？

> 速记层
> Agent 负责在不完整信息下选择步骤，确定性代码负责权限、校验、状态机和副作用。可靠性来自有限工具、结构化状态、预算、幂等、可观测性和评测集，而不是更长 Prompt。高风险操作让用户看到精确预览，并把确认绑定身份、参数摘要和期限。

1. 把工作流显式化

```
from dataclasses import dataclass
from enum import StrEnum


class Stage(StrEnum):
    COLLECT = "collect"
    ANALYZE = "analyze"
    REVIEW = "review"
    PUBLISH = "publish"
    FAILED = "failed"


@dataclass
class RunState:
    run_id: str
    stage: Stage
    attempts: int = 0
    artifact_id: str | None = None
    error_code: str | None = None
```

模型可以提出“下一步调用哪个只读工具”，但状态转换由代码验证：没有分析制品不能发布，没有确认不能进入 PUBLISH，失败不能通过自然语言声称完成。

2. 错误分类

至少区分：

- model_selection_error：选错工具或参数语义错误；
- validation_error：结构或领域约束失败；
- authorization_error：身份无权；
- tool_business_error：资源不存在、冲突或状态不允许；
- dependency_error：数据库、第三方 API、模型服务不可用；
- orchestration_error：状态丢失、循环、预算耗尽；
- evaluation_failure：流程完成但答案不满足质量标准。

分类决定重试策略。权限错误不能重试；限流要按 Retry-After；模型选错可在有限反馈后再规划；未知副作用结果必须先查状态。

3. 预算和停止条件

每个 Run 限制最大步骤、总耗时、模型调用次数、Tool 次数、输出字节和费用。重复相同调用或状态无进展触发停止。降级可以是返回部分结果和缺失原因，而不是无限循环。

4. 离线与在线评测

离线集合包含正常、边界、对抗和历史事故案例；每条记录输入、允许工具、期望关键事实、禁止动作和评分规则。除了最终答案，还评：工具选择、参数、权限遵循、步骤数、延迟、费用和引用证据。

在线监控成功率、人工否决率、空转率、工具错误、P95 延迟、单次成本和按版本回归。用户满意度是重要信号，但不能替代安全约束。

```
def exact_metric_score(actual: dict[str, float], expected: dict[str, float]) -> float:
    if actual.keys() != expected.keys():
        return 0.0
    passed = sum(abs(actual[key] - value) < 1e-9 for key, value in expected.items())
    return passed / len(expected) if expected else 1.0
```

数值和状态优先用确定性评分；开放文本再用规则、人工抽样或模型评审，并校准评审偏差。

5. 人工确认

确认页面显示工具名称、目标资源、变更前后、费用 / 影响范围和不可逆性。确认对象绑定 `actor_id + input_hash + policy_version + expires_at`。用户修改参数后必须重新确认。

> 深挖层：可恢复执行
> 把每个有副作用步骤写成可重入状态机：先保存意图和幂等键，再调用外部系统，最后保存结果。进程崩溃后根据外部操作 ID 对账。仅把消息历史存下来，不能证明业务动作处于什么状态。

> 边界案例：模型说“报告已发送”
> 只有发送 Tool 的结构化结果和外部消息 ID 能证明提交；最终送达还可能需要回执。Agent 的自然语言不是事实源。若 Tool 超时，答案应写“状态未知，正在核对”，不能补全成成功。

6. 项目与面试

风险报告 Agent 的 MVP 固定四阶段：读取授权数据、计算指标、生成草稿、人工确认后发布。先建立 30-50 条评测案例，再比较 Prompt 或模型版本；否则每次优化都只是在 Demo 上凭感觉。

问：如何降低幻觉？

缩小任务、提供权威 Resource、用结构化 Tool 获取事实、强制关键结论引用证据、对数字做程序校验、给未知状态明确表达，并用回归评测监控。不能承诺彻底消除概率错误。

> 练习
>
> 1. 为四阶段风险 Agent 写合法和非法状态转换。

2. 设计 10 条包含越权、空数据、超时和重复调用的评测案例。
3. 为发布动作设计确认令牌和崩溃恢复流程。

官方参考：MCP Security Best Practices；MCP Elicitation；MCP Tasks Extension

D06 MCP 测试、安全与部署

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 28 分钟

先闭卷回答

1. MCP Server 的测试金字塔应包含什么？
2. 为什么 token passthrough 会破坏安全边界？
3. 本地 stdio 与远程 HTTP 各有哪些专属风险？
4. 新旧协议同时服务时，水平扩展要验证什么？

速记层
先单测普通业务服务，再用进程内 Client 测工具契约，最后覆盖真实 stdio / HTTP、认证、断流和目标 Host。远程 Server 是独立资源服务器：验证令牌受众、权限和调用者；不把上游 Token 原样转发。部署要限制 Host / Origin、请求大小、并发和出站网络，并区分 2026 无会话路径与旧会话路径。

1. 测试分层

层级	验证内容	常见遗漏
领域单测	权限、计算、不变量、错误分类	只测 happy path
进程内契约	Schema、发现、序列化、结构化结果	把框架内部当业务测试
传输集成	stdio 帧、HTTP Header、超时、断流	只用内存 Client
兼容测试	新旧协议、目标 Host、SDK 版本	只测 Inspector
安全测试	越权、注入、SSRF、资源耗尽、审计	只测认证成功

```
import pytest

from mcp import Client


@pytest.mark.anyio
async def test_tool_returns_structured_result(mcp_server) -> None:
    async with Client(mcp_server) as client:
        result = await client.call_tool(
            "summarize_risk",
            {"project_id": "P-42", "threshold": 0.8},
        )
    assert result.structured_content["project_id"] == "P-42"
    assert result.structured_content["level"] in {"normal", "high"}
```

测试用例不要只断言文本包含“成功”，要断言字段、错误码、副作用记录和审计 ID。Schema 快照可提示破坏性变更，但更新快照前必须人工审查语义差异。

2. 身份与授权

远程 HTTP Server 校验发行者、签名、受众、过期时间和 Scope，再在每个资源上做业务授权。客户端给 MCP Server 的 Token 只应代表调用 MCP 资源；Server 调用下游服务时使用正规 OAuth 交换或自己的受限凭证。

Token passthrough 风险：Server 不验证、直接把客户端 Token 转给下游，会让受众和审计混乱，下游无法区分调用链，也可能把高权限 Token 暴露给不该看到的服务。

3. 输入、输出与出站网络

- 所有 Tool 参数按不可信输入处理；文件路径做规范化并限制根目录。
- URL 获取采用协议、域名和端口白名单，解析后阻止回环、链路本地和内网地址；重定向后再次校验。
- 限制请求体、文件、返回条目、递归深度、执行时间和并发。
- 日志不记录 Token、Cookie、Prompt 中的秘密或完整敏感结果。
- 高权限 Server 使用独立进程 / 容器、最小文件权限和受限出站网络。

stdio Server 还要防止当前工作目录、环境变量和 PATH 注入；安装配置中的命令和参数本质上等同本地代码执行授权。

4. HTTP 边界

本地 HTTP Server 应验证 Host，并按规范和部署方式限制 Origin，以降低 DNS rebinding。生产入口使用 TLS，认证失败不泄露资源存在性。网关超时、应用超时和下游超时形成总预算，不能每层各等 60 秒。

5. 扩展与兼容部署

2026-07-28 请求本身无会话，普通负载均衡即可分发；多轮 request_state 的加密 / 签名密钥要在副本间一致。旧协议客户端仍可能依赖 Session ID、进程内会话和粘性。跨副本列表变更通知需要共享事件总线，不能只在当前 Worker 发布。

> 深挖层：兼容矩阵比“支持 MCP”更诚实
> 记录 Server SDK、协议版本、传输、认证方式、Host 版本和功能原语。每次升级跑矩阵。协议符合不代表某 Host 会展示所有 Resource、支持 Tasks 或接受相同认证流程。

> 边界案例：stdio 日志破坏协议
> 一行 `print("connected")` 混入 stdout 可能让 Client 解析失败，而在终端手测看似正常。统一日志到 stderr，并在子进程集成测试中验证 stdout 只有协议帧。

6. 上线清单与面试

上线前至少验证：最小权限、密钥轮换、审计、限流、超时、优雅退出、健康检查、版本兼容、Schema 变更、敏感输出、断流重试和幂等。Inspector 用于开发探索，不是完整安全认证。

问：MCP Server 如何做水平扩展？

先说明目标协议。新协议普通请求无会话，但 request_state 密钥、通知总线、业务幂等和持久任务仍需共

享；旧客户端可能要求粘性或共享会话。数据库连接池总量还要乘以副本数控制。

> 练习
>
> 1. 为风险 Tool 写未认证、跨租户、超大时间窗和重复幂等键测试。
> 2. 用真实子进程验证 stdout 纯净、stderr 有结构化日志。
> 3. 列出一个两副本部署的新旧协议兼容矩阵。

官方参考：MCP Python SDK；Authorization；Security Best Practices

第五篇：部署、性能与安全

E01 从指标到 Profile 的性能优化

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 平均响应时间为什么会掩盖真实问题？
2. CPU Profile 与采样 Profile 各适合什么场景？
3. 如何判断优化的是代码、SQL、网络还是排队？
4. 微基准结果为什么不能直接代表生产吞吐？

> 速记层
> 先定义用户可感知目标，再用分位数、吞吐、错误率和资源饱和度定位。先测量后 Profile；先改算法、I/O 次数和数据量，再考虑局部语法。基准固定输入、环境和预热，多次运行并报告分布。优化后必须回归正确性与生产指标。

1. 建立性能问题陈述

一个可行动的问题应写成：“生产 API `/reports` 在 100 并发、30 天查询范围下 P95 从 420 ms 升到 1.8 s，数据库 CPU 85%，错误率未变。”它比“Python 太慢”包含负载、分位数、时间和资源信号。

常用四类信号：

- Latency：P50、P95、P99，区分排队和执行。
- Traffic：QPS、任务到达率、数据规模。
- Errors：超时、5xx、业务拒绝、降级。
- Saturation：CPU、内存、连接池、线程池、队列、磁盘和网络。

平均值会把少数极慢请求稀释；P99 也可能因样本过少而抖动。必须同时给样本数和时间窗。

2. 正确使用计时

```
from time import perf_counter


def timed_call(function, *args, **kwargs):
    started = perf_counter()
    try:
        return function(*args, **kwargs)
    finally:
        elapsed = perf_counter() - started
        print(f"elapsed={elapsed:.6f}s")
```

`perf_counter()` 适合持续时间，不能当业务时间戳。微基准用 `timeit` 隔离重复运行，但要防止被测操作与真实输入、缓存状态、网络和并发完全不同。

3. Profile 工具选择

```
import cProfile
import pstats


def build_report() -> None:
    sum(value * value for value in range(100_000))


profiler = cProfile.Profile()
profiler.enable()
build_report()
profiler.disable()
pstats.Stats(profiler).sort_stats("cumtime").print_stats(15)
```

- `cProfile` 是确定性函数调用 Profile，适合复现环境；关注 cumulative time 与 internal time 的差别。
- 采样 Profiler 周期性观察栈，对线上侵入更小，但短函数可能漏采样。
- `tracemalloc` 观察 Python 分配来源；系统 RSS 还包含原生库、映射和分配器保留。
- 数据库慢查询、EXPLAIN 与分布式 Trace 用于跨进程 I/O，不能只看 Python 栈。

4. 常见高收益顺序

1. 删除不必要工作和重复请求；
2. 降低算法复杂度，例如 list 查找改 set / dict；
3. 批量 I/O，避免 N+1；
4. 缩小读取列、行和序列化体积；
5. 增加正确索引或缓存；
6. 并发隐藏 I/O 等待；
7. 最后才是局部循环和对象创建优化。

缓存必须定义一致性、失效、容量和穿透保护，否则只是把延迟问题换成正确性问题。

5. 并发与吞吐

线程适合阻塞 I/O；进程适合可分割 CPU 工作但有序列化和内存成本；asyncio 适合大量协作式 I/O。传统 GIL 构建下线程不能让纯 Python CPU 循环线性并行。自由线程构建也不意味着现有扩展、锁竞争和共享数据自动安全。

Little's Law 的工程直觉是：在稳定系统中，并发在途量约等于到达率乘平均停留时间。下游变慢会增加在途请求；没有队列上限和背压时，延迟与内存会一起恶化。

> 深挖层：Profile 看到的是当时负载的样本
> 函数占比会随数据、缓存、并发和调用路径改变。一次 Profile 不能证明永久热点。保存输入规模、commit、Python 构建、依赖、CPU 限额和原始 Profile，优化前后用相同条件对比。

> 边界案例：P95 下降但错误率上升
> 如果慢请求被更早超时，存活请求的 P95 会变好，却牺牲成功率。性能验收必须同时看吞吐、错误、超时和业务完成率，不能只选一个好看的指标。

6. 项目与面试

对风险报告接口建立 1、10、100 项目三档数据集，记录 Python 时间、SQL 时间、序列化时间和总耗时。优化前保存 EXPLAIN 与 Profile；优化后用集成测试验证指标数值不变。

问：如何优化一个慢 Python API？

先确认生产指标和复现负载，再用 Trace 拆出排队、应用、数据库和下游耗时；在最大贡献层做 Profile。优先减数据与 I/O，改后做同负载对比、正确性回归和灰度监控。

> 练习
>
> 1. 为一个慢接口写出包含分位数、并发和数据规模的问题陈述。
> 2. 比较 list 与 set 成员查找，但说明微基准不覆盖的生产因素。
> 3. 用 Profile 找出 CPU 热点并记录优化前后证据。

官方参考：Python Profilers；timeit；perf_counter

E02 内存、GC 与资源泄漏

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. Python 对象已释放时，RSS 为什么不一定下降？
2. 引用计数和循环 GC 分别处理什么？
3. 内存泄漏与无界缓存怎样区分？
4. 文件描述符泄漏为何可能表现成网络故障？

> 速记层
> 先分清 Python 对象增长、原生内存、分配器保留、缓存和文件描述符泄漏。CPython 主要靠引用计数及时回收，循环 GC 处理可达性循环；RSS 不等于活跃 Python 对象。用 tracemalloc 比较快照，用系统指标观察 RSS、FD、连接和队列，并修复所有权与上限。

1. 所有权决定生命周期

常见保留链：全局 dict、无界 LRU、回调列表、未完成 Task、ThreadLocal、异常 traceback、队列和 ORM Session。对象不是“忘记 free”，而是仍被某条强引用链持有。

```
from functools import lru_cache


@lru_cache(maxsize=1024)
def normalized_project_name(project_id: str) -> str:
    return project_id.strip().upper()
```

有界缓存只是限制数量；若单个 value 很大、Key 高基数或数据有时效，还需按字节容量、TTL 和主动失效设计。

2. 引用计数与循环 GC

CPython 引用计数归零时通常立即析构；互相引用的对象即使业务上不可达，计数仍非零，由分代循环 GC 检测。GC 不是“定时释放所有内存”，也不负责关闭仍被引用的 Socket。

```
class Node:
    def __init__(self) -> None:
        self.peer: "Node | None" = None


left = Node()
right = Node()
left.peer = right
right.peer = left
```

退出作用域后循环可能等待 GC。更重要的是先判断为何需要双向强引用；可用明确 `close()`、弱引用或单向所有权简化。

3. tracemalloc 快照

```
import tracemalloc


tracemalloc.start(25)
before = tracemalloc.take_snapshot()

objects = [str(number) for number in range(100_000)]

after = tracemalloc.take_snapshot()
for stat in after.compare_to(before, "lineno")[:10]:
    print(stat)
```

比较稳定负载前后快照，并多轮确认持续增长。tracemalloc 追踪 Python 分配，不覆盖所有 C 扩展和内核资源。RSS 增长而 tracemalloc 平稳时，检查原生库、mmap、子进程、分配器碎片和容器统计。

4. 非内存资源

```
from pathlib import Path


def load_template(path: Path) -> str:
    with path.open(encoding="utf-8") as handle:
        return handle.read()
```

文件、Socket、数据库游标、临时目录和锁使用 context manager。连接池必须设置容量、获取超时和归还路径；流式响应在客户端断开时也应关闭生成器和下游连接。

监控进程打开 FD 数、连接池 in-use / wait、线程、Task、队列长度和临时文件。达到 FD 上限后，新连接、DNS、文件读取都可能报“Too many open files”，看起来像多个系统同时坏了。

5. 背压与大对象

批量任务应流式处理或分块，不能先把全部数据库行、全部 JSON 和最终压缩包同时放内存。生产者速度高于消费者时使用有界队列；满时阻塞、拒绝或降级，而不是继续积压。

> 深挖层：RSS 不回落
> 对象释放后，CPython 的 pymalloc arena 或系统 malloc 可能保留页面供进程复用，未立即归还 OS；碎

片也会让 RSS 高水位持续。判断泄漏要看活跃对象 / 分配趋势和重复负载后的平台期，而不是一次 GC 后 RSS 是否下降。

边界案例：后台 Task 捕获大对象
闭包引用请求体并创建长期 Task，即使请求结束，大对象仍存活。任务应只接收最小标识，从持久化层读取需要的数据，并有取消、超时和完成清理。

6. 项目与面试

对 10,000 行风险导出做循环压力测试：记录每轮对象快照、RSS、FD 和连接池。若 RSS 上升但对象平台稳定，继续看压缩库和分配器；若某行对象数持续增长，沿引用链查全局容器或未完成任务。

问：如何排查 Python 内存泄漏？

先复现增长曲线并区分 RSS、Python heap 和资源数；用 tracemalloc 快照定位分配，用 gc / 对象图确认引用链，再检查缓存、任务、Traceback 和 C 扩展。修复后做多轮稳态验证。

练习

1. 构造一个无界缓存增长，再加入容量、TTL 与指标。
2. 故意遗漏文件关闭，观察 FD 而不仅是内存。
3. 比较一次大峰值与每轮线性增长的诊断结论。

官方参考：tracemalloc；gc；weakref

E03 Linux 进程、信号与服务运行

速记复习 10-15 分钟；完整阅读约 14 分钟；深挖约 24 分钟

先闭卷回答

1. 应用为什么不应直接用 Flask 开发服务器上线？
2. SIGTERM 到来时，进程应按什么顺序退出？
3. readiness 与 liveness 有什么不同？
4. 多 Worker 的数据库连接数如何计算？

速记层
生产服务由进程管理器启动、重启、传递信号并收集日志；应用服务器承接 HTTP。SIGTERM 后停止接收新工作、等待有界在途任务、持久化状态、关闭连接，再退出。liveness 判断是否需重启，readiness 判断是否接流量。容量按副本 × Worker × 每 Worker 连接池计算。

1. 进程模型

Flask 开发服务器为开发调试设计。生产可用 WSGI Server 托管 Flask，由 systemd、容器平台或其他管理器负责进程生命周期。Worker 数量不是 CPU 核数的固定公式，要通过负载测试结合 I/O、内存和连接池调节。

pre-fork 模型中，主进程监听并管理 Worker。fork 前创建的线程、Socket、数据库连接通常不能安全地在子

进程共享；连接池应在 Worker 内建立。fork 的 Copy-on-Write 能共享未修改页面，但写入会复制。

2. 信号与优雅退出

```
import signal
import threading


stopping = threading.Event()


def request_shutdown(signum: int, frame: object) -> None:
    stopping.set()


signal.signal(signal.SIGTERM, request_shutdown)
signal.signal(signal.SIGINT, request_shutdown)
```

Signal Handler 保持极简，只设置标志；主循环或服务器生命周期执行清理。退出步骤：标记 not-ready、停止领取新任务、等待在途到 deadline、取消可取消工作、归还租约 / 保存检查点、刷新必要日志、关闭池。

如果平台 30 秒后 SIGKILL，而应用优雅超时设为 60 秒，清理永远不会完成。各层 deadline 要留出余量。

3. systemd 单元要点

```
[Unit]
Description=Risk API
After=network-online.target

[Service]
Type=simple
User=risk-api
WorkingDirectory=/srv/risk-api
EnvironmentFile=/etc/risk-api/env
ExecStart=/srv/risk-api/.venv/bin/gunicorn "app:create_app()"
Restart=on-failure
TimeoutStopSec=35
NoNewPrivileges=true
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

密钥文件权限只给服务用户；不要把 Secret 写在 Unit 或命令行。`Restart=always` 会把配置错误变成高速崩溃循环，必须配合退避、启动限制和告警。

4. 健康检查

- liveness：进程事件循环是否还能执行。失败触发重启，检查应轻量。
- readiness：是否应接收新请求，例如正在排空或关键依赖不可用。
- startup：冷启动、迁移或模型加载需要更长时间时避免误杀。

不要让 liveness 强依赖外部数据库，否则数据库故障会让所有应用副本同时重启，加剧雪崩。readiness 是否检查依赖取决于是否还有降级能力。

5. 日志与权限

容器 / systemd 环境优先把结构化日志写 stdout/stderr，由平台收集；应用不自行轮转同一文件。服务使用

非 root 用户，只读代码和配置，写入仅限明确数据目录。

> 深挖层：退出期间的竞态
> 标记 not-ready 到负载均衡停止发流量有传播延迟。应用先拒绝或短暂保留接收能力、再 drain；精确顺序取决于平台。后台任务使用租约和幂等，使进程被 SIGKILL 也能由其他 Worker 恢复。

> 边界案例：Worker 数翻倍
> 4 副本 × 4 Worker × 每 Worker 10 个数据库连接可能产生 160 个连接，还未计后台任务和管理连接。应用扩容前必须核对数据库上限并给连接池等待设置超时。

6. 项目与面试

为 Flask 风险 API 提供 `create_app()` 工厂、生产 WSGI 配置、readiness、SIGTERM 排空和 systemd / 容器运行方式。演示滚动部署期间长查询不丢失，并说明超时后如何恢复。

问：线上进程收到 SIGTERM 应做什么？

立即停止接新工作或标记不就绪，在平台期限内完成 / 取消在途任务，保存可恢复状态，关闭连接和资源，再以正常状态退出；关键副作用由幂等和持久状态保障，不能只靠 finally。

> 练习
>
> 1. 计算当前服务最大数据库连接数并留管理余量。
> 2. 写一个 10 秒任务，在 SIGTERM 下验证 drain 与强杀边界。
> 3. 设计 liveness、readiness 和 startup 的不同返回条件。

官方参考：Flask Production Deployment；signal；systemd.service

E04 依赖、配置、容器与可回滚发布

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 26 分钟

先闭卷回答

1. `pyproject.toml`、锁文件和构建制品分别解决什么？
2. 为什么只写 `package>=1` 不能复现部署？
3. 数据库迁移为何要与应用回滚协同？
4. 容器镜像小是否就意味着安全？

> 速记层
> 项目元数据声明直接依赖，锁文件固定完整解析，Wheel / 镜像是可发布制品。配置有明确优先级并启动时校验，Secret 不入仓库和镜像。镜像固定基础版本、非 root、最小运行依赖并生成可追踪版本。发布采用向后兼容迁移、健康验证、灰度和一键回滚。

1. 三类依赖信息

信息	作用	边界
`pyproject.toml`	项目、Python 版本、直接依赖与构建系统	范围不一定固定间接依赖
lock 文件	在目标平台固定解析版本和哈希	需随依赖变更审查更新
Wheel / 镜像	经过测试的交付制品	仍需来源、签名和漏洞治理

生产不要现场从浮动范围重新解析并构建。CI 构建一次不可变制品，在测试、预发和生产逐级提升；发布记录 commit、依赖锁摘要、镜像 digest、配置版本和迁移版本。

2. 配置模型

```
from dataclasses import dataclass
import os


@dataclass(frozen=True)
class Settings:
    database_url: str
    request_timeout_seconds: float
    environment: str

    @classmethod
    def from_env(cls) -> "Settings":
        timeout = float(os.environ.get("APP_REQUEST_TIMEOUT", "10"))
        if not 0.1 <= timeout <= 60:
            raise ValueError("APP_REQUEST_TIMEOUT must be in [0.1, 60]")
        return cls(
            database_url=os.environ["APP_DATABASE_URL"],
            request_timeout_seconds=timeout,
            environment=os.environ.get("APP_ENVIRONMENT", "development"),
        )
```

启动时 Fail Fast，错误指出变量名但不打印 Secret。优先级例如：代码安全默认值 < 配置文件 < 环境变量 < 命令行；团队必须固定，避免不同入口行为不同。

3. 容器构建原则

```
FROM python:3.14-slim AS builder
WORKDIR /build
COPY pyproject.toml uv.lock ./
RUN pip install --no-cache-dir uv && uv sync --frozen --no-dev

FROM python:3.14-slim
RUN useradd --create-home --uid 10001 appuser
WORKDIR /app
COPY --from=builder /build/.venv /app/.venv
COPY src ./src
USER appuser
ENV PATH="/app/.venv/bin:$PATH"
CMD ["python", "-m", "risk_api"]
```

这是结构示例，实际要确保两个阶段 ABI 一致、锁文件支持目标平台，并固定基础镜像 digest。不要把 `.env`、SSH Key、包仓库 Token 或构建缓存复制进镜像层。多阶段减少工具链攻击面，但基础镜像、应用依赖和运行权限仍需扫描。

4. 数据库迁移

滚动发布中旧版和新版会同时运行。安全的 expand / migrate / contract：

1. expand：增加兼容字段 / 表，不删除旧结构；
2. deploy：新版双读或双写，处理旧数据；
3. migrate：后台回填并对账；
4. switch：切换读取路径并观察；
5. contract：确认无旧实例后删除旧结构。

一个事务内重命名列再立刻发布新代码，回滚旧代码可能无法启动。大表 DDL 还要评估锁、复制延迟和磁盘。

5. 发布与回滚

上线前跑单元 / 集成 / 迁移 / 安全测试；灰度关注错误、延迟和业务指标。回滚优先复用上一不可变制品，而不是临时重构建。若迁移不可逆，应用回滚不等于数据回滚，需要向前修复或兼容开关。

> 深挖层：可复现不等于跨平台相同
> 锁定版本仍可能因操作系统、CPU、Python ABI、系统库和可选依赖得到不同 Wheel。CI 应在与生产一致的平台构建，并保存制品哈希；不要在开发机锁定后假设 Linux 容器一定可安装。

> 边界案例：Secret 已从 Git 删除
> 历史 commit、构建日志、镜像层和缓存仍可能包含它。应立即吊销 / 轮换，再清理历史与制品；仅删除当前文件不能恢复机密性。

6. 项目与面试

风险平台交付清单包含：固定依赖、非 root 镜像、SBOM / 扫描结果、配置 Schema、迁移计划、灰度阈值和回滚演练。README 写清从 commit 到制品的可追踪链。

问：怎样实现零停机数据库发布？

承认严格“零”取决于平台和业务；核心是新旧版本兼容、先扩后缩、后台回填、短事务、流量灰度和可回滚制品。对高风险 DDL 做影子演练和容量评估。

> 练习
>
> 1. 为删除一个列写 expand / migrate / contract 计划。
> 2. 检查镜像中是否含开发依赖、root 用户和 Secret。
> 3. 设计一次配置错误的启动失败与回滚验证。

官方参考：Writing pyproject.toml；Docker Multi-stage Builds；Python Build Configuration

E05 日志、指标、Trace 与健康度

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 24 分钟

先闭卷回答

1. 日志、指标和 Trace 分别最擅长回答什么？
2. request_id 与 trace_id 有何不同？
3. 为什么把 user_id 放进指标标签可能出事故？
4. 健康检查通过为何仍可能无法服务用户？

> 速记层
> 日志解释离散事件，指标展示总体趋势并告警，Trace 串联一次请求的跨服务路径。统一传播 trace context 和 request_id，结构化记录稳定字段。指标标签必须低基数；告警围绕用户影响和 SLO。健康检查只是流量 / 重启信号，不等同完整业务验证。

1. 三根支柱的分工

信号	擅长问题	不适合
Log	发生了什么、错误上下文、审计事件	聚合高频趋势
Metric	多少、趋势、分位数、是否告警	单次请求完整细节
Trace	时间花在哪、跨服务因果路径	长期精确计费或审计

三者通过 service、environment、version、trace_id 和 request_id 关联，但不要把敏感 payload 全量复制到每个信号。

2. 结构化日志

```
import logging


logger = logging.getLogger("risk_api")


def load_report(report_id: str, request_id: str) -> None:
    logger.info(
        "report_load_started",
        extra={"report_id": report_id, "request_id": request_id},
    )
```

生产 Formatter 输出 JSON；事件名稳定，字段单独存储，避免后续正则解析自然语言。异常只在能添加上下文或决定处理的边界记录一次，防止同一错误每层重复打印。

日志级别：DEBUG 用诊断且通常关闭；INFO 记录关键状态；WARNING 表示已降级 / 可恢复异常；ERROR 表示请求或任务失败。不要把用户输错参数全记成 ERROR。

3. 指标与基数

Counter 适合请求数和错误数；Gauge 适合队列深度与池使用量；Histogram 适合延迟和大小。标签可用 route 模板、method、status_class、error_code；不要用完整 URL、request_id、user_id 或异常文本，它们会制造高基数和成本爆炸。

客户端和服务器 Histogram 桶、单位要一致。平均延迟可从 sum / count 得到，但仍需分位数与分布。

4. Trace 与上下文传播

入口提取或创建 Trace Context，为数据库、HTTP、队列、模型和 MCP Tool 建 Span。异步消息把 traceparent 作为元数据传播；消费者创建新的处理 Span，并用业务 job_id 补充关联。

Span 属性不记录 Secret、原始 SQL 参数或整段 Prompt。采样会漏掉多数正常请求，错误和高延迟可提高采样，但 Trace 不能承担完整审计。

5. SLI、SLO 与告警

SLI 是测量，例如“有效请求中 2 秒内成功的比例”；SLO 是目标，例如 30 天 99.9%。告警优先基于错误预算消耗和用户影响，资源使用告警作为诊断辅助。

只有 CPU 90% 不一定有用户影响；CPU 30% 也可能因单个锁或下游超时全站变慢。Dashboard 同屏展示流量、错误、延迟、饱和度和版本发布标记。

> 深挖层：Context 传播与业务关联
> trace_id 描述一次技术调用链，request_id 可由网关生成并用于外部支持，job_id / idempotency_key 描述可跨多次请求的业务执行。它们可以关联但不能互相替代。

> 边界案例：健康接口永远返回 200
> 进程活着但线程池耗尽、连接池等待或配置版本错误，简单 liveness 仍通过。readiness、容量指标和合成业务探测共同覆盖；合成探测也要用隔离账号和可清理数据。

6. 项目与面试

风险报告 Run 记录 run_id、trace_id、commit、输入摘要、阶段、耗时、Tool 调用、行数和制品 ID。Dashboard 展示各阶段 P95、失败分类和队列等待；失败页面从 run_id 跳到 Trace 和脱敏日志。

问：出现偶发慢请求怎样排查？

先按时间、版本、路由和租户特征定位指标，再从慢样本 Trace 拆分排队、SQL、外部调用和序列化；用关联日志看错误上下文。若未采样，检查同窗口资源与慢查询，并调整后续采样策略。

> 练习
>
> 1. 把一条拼接日志改成稳定事件名和结构化字段。
> 2. 从指标标签中移除高基数字段并保留排查能力。
> 3. 为报告生成定义 SLI、SLO 和两档告警。

官方参考：Python Logging HOWTO；OpenTelemetry Signals；W3C Trace Context

E06 Python 应用安全边界

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 27 分钟

先闭卷回答

1. 认证通过后为什么仍会发生越权？
2. `subprocess.run(..., shell=False)` 还要校验什么？
3. SSRF 为什么不能只拦截字符串 `localhost`？
4. 哪些 Python 反序列化方式不能处理不可信数据？

> 速记层
> 安全从数据流和权限边界分析：谁控制输入、代码以谁的权限运行、数据去哪里。所有入口做结构与业务校验；每个对象做授权；SQL 参数绑定；命令不用 Shell；路径限制根目录；出站 URL 防 SSRF；不对不可信数据使用 pickle。Secret 最小化、轮换、脱敏，依赖和镜像持续治理。

1. 认证、授权与对象边界

认证回答“是谁”，授权回答“能否对这个对象做这个动作”。仅验证 `project_id` 是 UUID 不能防 IDOR；必须在查询中绑定可信 actor 的租户 / 项目范围。

```
def get_project(repository, actor_id: str, project_id: str):
    project = repository.find_visible_project(actor_id, project_id)
    if project is None:
        raise LookupError("project not found")
    return project
```

对无权与不存在返回相同外部语义可减少枚举；内部审计仍区分。管理员能力单独建模，不能用客户端传 `is_admin=true`。

2. 注入与解释器边界

- SQL：值参数绑定；动态列 / 排序白名单。
- Shell：优先调用库；必须启动进程时传参数列表、`shell=False`、固定可执行文件绝对路径和受限环境。
- 模板：使用自动转义，禁止把用户文本标记为 safe。
- 日志：控制换行和结构字段，避免日志伪造；不得写凭证。
- Prompt：外部文档中的指令是数据，不是权限；Tool 服务端仍做授权。

```
import subprocess


def convert_pdf(source: str, destination: str) -> None:
    subprocess.run(
        ["/usr/bin/pdftotext", "--", source, destination],
        check=True,
        timeout=20,
        env={"PATH": "/usr/bin:/bin"},
    )
```

即使不用 Shell，也要限制路径、文件大小、可执行程序行为和超时。攻击者仍可能传特殊文件、消耗资源或利用被调用程序漏洞。

3. 路径与上传

```
from pathlib import Path


UPLOAD_ROOT = Path("/srv/app/uploads").resolve()


def safe_upload_path(name: str) -> Path:
    candidate = (UPLOAD_ROOT / name).resolve()
    if UPLOAD_ROOT not in candidate.parents:
        raise ValueError("path escapes upload root")
    return candidate
```

还需处理根目录本身、符号链接竞态、文件覆盖、保留名、扩展与真实内容不符。更稳是服务端生成随机文件名，把原名只作元数据，在隔离位置扫描后再发布。

4. SSRF

URL 白名单至少检查 scheme、认证信息、主机、端口；DNS 解析后拒绝 loopback、private、link-local、保留地址；连接时防 DNS rebinding；每次重定向重新验证。HTTP Client 禁止自动读取云元数据地址，并限制响应大小与超时。

只匹配 `localhost` 会漏掉 `127.0.0.1`、IPv6、整数 IP、DNS 指向内网和重定向。

5. 反序列化与动态执行

不对不可信输入使用 `pickle.loads`、`marshal` 或任意对象反序列化；YAML 使用安全加载并限制类型；避免 `eval`、`exec`。JSON 本身不执行代码，但仍需限制深度、大小和字段，再做 Schema 与业务校验。

签名只能证明数据来自持钥者和未被修改；若签名者可能生成恶意 pickle，反序列化仍执行代码。优先无代码执行语义的数据格式。

6. Web 基线与供应链

Cookie 设置 Secure、HttpOnly、合适 SameSite；状态修改请求防 CSRF；输出编码防 XSS；登录、重置和敏感操作限流。密码使用专用慢哈希算法，绝不能明文或通用快速 Hash。

依赖固定并持续扫描，审查名称混淆、维护者变更和安装脚本。Secret 通过受控存储注入，支持轮换；日志、Trace、错误页、测试制品和 AI Prompt 都是潜在泄露面。

> 深挖层：最小权限要贯穿调用链
> Web 用户、应用进程、数据库账号、对象存储、MCP Server 和下游 API 各有独立身份与最小 Scope。共享一个全能 Token 虽省事，却让任何单点输入漏洞升级为全域权限。

> 边界案例：安全的文件扩展名
> `report.pdf` 可能不是 PDF，合法 PDF 也可能含恶意结构或巨大解压内容。校验魔数 / 解析结果、大小、页数和处理超时；转换器放隔离进程，不信任文件名。

7. 项目与面试

为风险平台做一张数据流威胁模型：浏览器、API、数据库、对象存储、模型和 MCP Server。标出身份、Secret、信任边界、输入验证、出站连接和审计，再用测试对应每个控制。

问：如何防止 Prompt Injection 导致数据泄露？

把文档指令当不可信数据；限制模型可见上下文和 Tool；服务端按真实身份授权；敏感动作人工确认；输出做数据策略检查；记录审计并用对抗案例评测。单靠系统 Prompt 不能成为权限控制。

> 练习
>
> 1. 为下载 URL 写含重定向和 DNS 解析的 SSRF 测试表。
> 2. 检查项目中 pickle、eval、shell=True 与不安全模板用法。
> 3. 为跨租户对象访问写授权失败测试。

官方参考：Python Security Considerations；Flask Web Security；OWASP Cheat Sheet Series

E07 故障定位、恢复与复盘

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 事故处理中为何先止损而不是先找完美根因？
2. 相关性与因果性如何区分？
3. 什么时候重启是合理恢复，什么时候会破坏证据？
4. 复盘中的“根因”为什么常常不止一个？

> 速记层
> 先确认用户影响和时间线，再止损、保留证据、形成假设、用可证伪实验定位。每次只改变一个关键变量。恢复手段应可回滚并监控。复盘关注触发条件、放大因素、检测与恢复缺口，用有负责人和期限的系统改进替代“以后注意”。

1. 事故前 15 分钟

1. 指定协调者并记录时间线；
2. 确认影响面：哪些用户、操作、区域、版本；
3. 检查最近发布、配置、依赖和流量变化；
4. 选择低风险止损：暂停发布、回滚、限流、关闭功能、切只读；
5. 保存日志、Trace、Profile、指标和关键状态；
6. 设定下一次同步时间。

不要在没有记录的情况下多人同时改生产。若正在持续破坏数据，止损优先于完整诊断；若系统稳定且问题难复现，先采证再重启。

2. 假设驱动排查

把“数据库有问题”改成可验证假设：“新查询缺索引导致扫描量增加，连接占用延长，使连接池等待成为 P95

主因。”证据包括 EXPLAIN、慢查询、池等待、版本差异和回滚对比。

同一时间发生不等于因果。发布后 CPU 上升可能因流量突增；需要比较对照组、时间顺序和机制。先用最便宜、信息增益最大的检查排除大类。

3. Python 现场诊断

```
import faulthandler
import signal


faulthandler.register(signal.SIGUSR1, all_threads=True)
```

在 Unix 上可用信号请求线程栈转储，具体信号和安全策略按平台配置。死锁排查关注所有线程等待的锁；asyncio 卡顿关注事件循环阻塞、未完成 Task 和同步 I/O。CPU 高用采样 Profile；内存高用 tracemalloc / heap 与 RSS 对照；FD 高查 `/proc` 和资源所有权。

核心转储和完整内存可能含 Secret，访问与保留要受控。线上动态诊断也有开销，先在单副本或采样窗口执行。

4. 常见故障模式

现象	候选方向	第一批证据
延迟升高、CPU 低	下游慢、锁、池等待、队列	Trace、连接池、线程栈
CPU 高	热循环、序列化、重试风暴	采样 Profile、流量、版本
RSS 持续升	对象保留、原生分配、队列	tracemalloc、RSS、队列
大量 502 / 504	Worker 崩溃、超时层次、网关	进程退出、网关与应用日志
数据重复	超时重试、幂等缺失、重复消费	幂等记录、消息投递、审计

5. 恢复与验证

回滚应用前确认数据库结构仍兼容；切流前确认副本数据延迟；清缓存前评估回源洪峰；扩容前检查下游连接容量。每个动作定义预期指标、观察时间和撤销方法。

恢复后验证用户路径和数据一致性，不只看进程绿色。对副作用未知的任务做对账：外部 ID、幂等记录和数据库状态一致后再决定补偿或重试。

6. 无责复盘但责任明确

复盘包含：影响、时间线、检测、触发、促成条件、缓解、恢复、做得好 / 不足和行动项。避免只写“开发遗漏测试”；追问为什么设计允许单点错误上线、为什么门禁没发现、为什么告警晚。

行动项具体到系统、负责人、期限和验收，例如“在 9 月 15 日前给所有写 Tool 增加幂等契约测试并纳入 CI”，而不是“加强稳定性”。

> 深挖层：失败链而非单一根因

一次事故常由触发事件 + 潜在缺陷 + 放大机制 + 检测缺口组成。缺索引触发慢查询，过大连接池放大数据库压力，无背压继续接流量，告警只看 CPU 导致发现晚。修一个点降低概率，修整条链提高韧性。

边界案例：回滚反而失败
新版已写入旧版不认识的数据或执行了破坏性迁移，代码回滚会再次报错。发布前必须设计向后兼容；事故中若已越过不可逆点，选择前向修复或兼容开关。

7. 项目与面试

为风险平台预写三份 Runbook：数据库连接池耗尽、报告任务重复执行、MCP Tool 超时未知。每份包含指标、查询、止损、验证、权限和升级联系人，定期演练。

问：讲一次你解决线上问题的经历。

按影响 - 证据 - 假设 - 动作 - 结果 - 预防表达，给出数字和职责边界。不要把“重启好了”当根因；说明为何恢复、如何确认数据、如何防复发。

练习

1. 为连接池耗尽写前 15 分钟 Runbook。
2. 把“服务挂了”改写成三个可证伪假设。
3. 给一次虚拟事故写带负责人和验收标准的行动项。

官方参考：faulthandler；tracemalloc；Google SRE Postmortem Culture

第六篇：项目表达与面试

F01 把知识变成可验证的项目证据

速记复习 10-15 分钟；完整阅读约 14 分钟；深挖约 22 分钟

先闭卷回答

1. “熟悉 Python”为什么不是有效项目证据？
2. 一个技术主张至少需要哪四部分？
3. 没有真实线上流量时，如何诚实证明性能？
4. README、测试和运行制品分别证明什么？

> 速记层
> 项目表达使用“问题 - 约束 - 决策 - 实现 - 证据 - 边界”链。每个能力落到可运行代码、自动测试、数据结果或故障演练。指标只写真实测量并说明环境；未上线就说负载测试，不包装成生产数据。面试官追问时，从结论下钻到机制、代码、失败模式和取舍。

1. 技术主张的四层证据

层级	示例	证明力
声明	“熟悉 pytest”	只说明关键词
实现	有 fixture、参数化和插件代码	证明做过功能
验证	有隔离、失败注入、并发和 CI 结果	证明理解边界
运行	有版本、指标、制品、复盘和演进记录	证明工程闭环

项目不是技术名词总和。选择少数关键难题，讲清为何选、怎样验证、哪里没做。

2. 一页项目卡

每个项目维护一页：

- 目标用户与痛点：谁在什么流程中浪费时间或承担风险；
- 范围：MVP 做什么，不做什么；
- 约束：数据权限、并发、成本、期限、团队规模；
- 架构：入口、服务、存储、异步任务和外部依赖；
- 两到三个关键决策：候选方案、选择理由、代价；
- 质量证据：测试、性能、安全、可观测和恢复；
- 结果：真实功能、测量结果和下一步。

3. 指标的可信写法

“性能提升 80%”缺少基准。至少说明：指标、前后值、样本、数据规模、并发、环境和改动。若只是实验：

> 在固定 100 项目、30 天数据集和 20 并发的本地容器环境中，比较优化前后 P95；该结果用于回归，不代表生产容量。

不要为了简历填数字而编造。没有生产数据时，可证明：自动化覆盖的业务路径数、故障注入通过项、负载测试分位数、静态 / 依赖扫描、重启恢复时间和对账结果。

4. 可运行证据包

建议仓库最小结构：

```
docs/
  architecture.md
  decisions/
  runbook.md
src/
tests/
pyproject.toml
README.md
```

README 给出 5 分钟启动、核心场景、架构图、测试命令、已知边界。Decision Record 写上下文、选项、决定、后果。CI 产出测试、覆盖、镜像、依赖清单和可追踪版本。

5. 从知识单元到项目证据

知识	证据动作
事务与锁	两并发事务测试 + 死锁 / 重试说明
pytest fixture	数据库隔离与 teardown 失败测试
asyncio	超时、取消和阻塞调用监控
MCP Tool	Schema 快照、跨租户拒绝、幂等测试
部署	SIGTERM 排空、迁移与回滚演练
安全	威胁模型、越权 / SSRF / 注入测试

> 深挖层：证据链必须可复查
> 一句结论应能追到代码 commit、测试输入、原始结果和环境。截图可以展示，但文本日志、报告和脚本更可复现。对关键性能结论保存 Profile / EXPLAIN，而不只保留最终数字。

> 边界案例：覆盖率 95%
> 高覆盖率可能来自大量无断言执行，也可能漏掉权限和并发。表达时说明覆盖口径、关键风险用例、Mutation / 故障注入或业务场景；覆盖率是缺口提示，不是质量结论。

6. 面试表达

两分钟结构：20 秒问题与职责，40 秒架构，40 秒关键难点，15 秒验证结果，5 秒边界。被追问时，不重复项目介绍，直接进入具体机制和一次失败案例。

问：你在项目中的最大贡献是什么？

用第一人称说明你拥有的决策和交付，同时区分团队成果。比如“我负责 Run / Task 状态机、租约恢复和对应并发测试；前端展示由另一位同事完成”。责任边界越清晰，可信度越高。

> 练习
>
> 1. 为一个项目写六段证据链，每段不超过两句。
> 2. 删除三个没有证据支撑的“精通 / 高性能 / 高可用”描述。
> 3. 录制两分钟介绍，标出所有可能被追问的数字和名词。

参考：PEP 8；Python Packaging User Guide；本书各单元的官方资料与练习证据链。

F02 多维风险分析工具：项目设计与讲述

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 26 分钟

先闭卷回答

1. “风险分数”如何做到可解释和可追踪？
2. 查询、计算、报告和发布为何要分阶段？
3. 权限应在哪几层落实？
4. 哪些结果可以缓存，缓存键包含什么？

> 速记层
> 风险工具的核心不是图表，而是口径版本、授权数据、可复现计算和证据链。架构拆成采集 / 查询、指标计算、规则评分、报告制品和发布。每次 Run 固定输入快照、规则版本与代码版本；MCP 只暴露受限领域工具，不暴露任意 SQL。

1. 项目目标与边界

目标：让项目负责人按时间、区域、类别等维度查看风险指标，生成可审查报告，并保留计算依据。MVP 不做“AI 自动决定风险处置”，模型只辅助总结和解释；最终数据、规则和发布由确定性系统控制。

核心非功能要求：

- 同一输入、规则和版本得到同一结果；
- 任何数值能追到源数据和口径；
- 用户只能读取授权项目；
- 大查询有上限、超时与取消；
- 发布动作需确认、幂等和审计。

2. 领域模型

```
from dataclasses import dataclass
from decimal import Decimal


@dataclass(frozen=True)
class RiskFactor:
    name: str
    value: Decimal
    weight: Decimal


@dataclass(frozen=True)
class RiskScore:
    total: Decimal
    level: str
    factors: tuple[RiskFactor, ...]
    rule_version: str


def calculate_score(factors: tuple[RiskFactor, ...], version: str) -> RiskScore:
    total = sum((item.value * item.weight for item in factors), Decimal("0"))
    level = "high" if total >= Decimal("0.75") else "normal"
    return RiskScore(total, level, factors, version)
```

生产规则应从受审配置加载并校验权重、范围和版本。使用 Decimal 说明精度意图；最终阈值、舍入和缺失值策略都属于口径。

3. 数据与运行模型

最小表：Project、RiskEvent、MetricDefinition、RuleVersion、AnalysisRun、RunEvidence、ReportArtifact、AuditEvent。AnalysisRun 保存：project、时间窗、数据快照 / as_of、规则版本、commit、状态、输入哈希和错误码。

查询索引围绕真实过滤与排序，例如 `(project_id, event_time, category)`；不能因字段多就为每个维度单建索引。用 EXPLAIN 和代表性数据验证。

4. 执行路径

1. API / MCP Handler 从认证上下文获得 actor；
2. Application Service 检查 actor 对 project 的权限；
3. Repository 读取受限数据和口径；
4. Calculator 做纯函数计算；
5. Run 持久化输入摘要、结果与证据；
6. Reporter 生成制品；
7. 用户确认后 Publisher 幂等发布。

纯 Calculator 可大量参数化测试；数据库契约测试确认过滤、时间边界和事务；端到端测试覆盖用户可见报告。

5. 缓存与异步

只缓存可由 `(project, permission_scope, data_version, rule_version, query)` 完整决定的只读结果。不能漏掉权限或版本。高成本报告使用后台任务，Run 状态 queued / running / reviewing / completed /

failed；Worker 用租约领取，超时后可恢复。

报告生成中途失败要保留可诊断证据，但不能让半成品成为“最终报告”。对象存储先写临时 Key，验证摘要后原子更新制品引用。

6. MCP 与 AI 层

- Resource：指标定义、规则解释、报告模板；
- Tool：查询受限指标、创建草稿、查询 Run 状态；
- Prompt：按组织模板审查风险摘要；
- 发布 Tool：预览和确认分开，接受幂等键。

模型输出不能改写原始数值。摘要引用 `metric_id` / `evidence_id`，渲染前程序核对引用存在且值一致。

> 深挖层：可解释不只是列出权重
> 还要保存输入值来源、缺失值处理、标准化方式、规则版本、舍入、阈值和运行时间。若源数据被修订，历史报告应继续解释当时快照，而不是悄悄按最新数据重算。

> 边界案例：同一事件迟到
> 事件在报告完成后补录。系统要定义 event_time 与 ingest_time、报告 as_of 和重算策略。没有这些语义，“同一天数据”也可能不同。

7. 两分钟项目讲述

“这个工具解决多来源风险数据口径不一致、报告不可追溯的问题。我把流程拆成授权查询、纯函数评分、带版本 Run 和报告发布；每个结果绑定数据时间、规则版本和证据。难点一是联合索引与查询上限，难点二是发布幂等和失败恢复，难点三是 MCP Tool 的最小权限。质量上用黄金数据、跨租户测试、并发租约测试和负载基准验证。AI 只做有证据引用的摘要，不参与权限和最终数值。”

这段是结构示范，实际表达应替换为你确实实现并能展示的部分。

> 练习
>
> 1. 为一个指标写完整口径和三个边界数据。
> 2. 画出 AnalysisRun 状态机，并设计 Worker 崩溃恢复测试。
> 3. 为 MCP 查询 Tool 写跨租户、超时和截断测试。

关联阅读：B06-B09 数据与可靠性；D03-D06 MCP 工程；E01、E05、E06 性能、观测与安全。

F03 自动化测试平台：项目设计与讲述

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 26 分钟

先闭卷回答

1. 平台如何把一次执行绑定到不可变环境？
2. Worker 崩溃后，旧 Worker 为什么不能继续回写？
3. 接口与 UI 用例怎样共享领域模型又保持执行差异？
4. Flaky 重跑为何不能覆盖首次结果？

> 速记层
> 平台价值在任务建模、隔离调度、证据与质量闭环，不是给 pytest 套页面。Run 固定 commit、依赖、配置和环境；Task 用租约与 fencing token 防重复 Worker 回写；执行器插件适配 pytest / Playwright；结果保留首次运行、重试和制品。

1. MVP 目标

用户选择仓库 commit、测试范围和环境，平台创建 Run，拆分 Task，调度 Worker，实时展示状态并收集 Allure、Trace、日志、截图。MVP 先支持单团队和受控环境，不急于多租户计费、智能选例和复杂设备农场。

2. 状态模型

```
from dataclasses import dataclass
from enum import StrEnum


class TaskState(StrEnum):
    QUEUED = "queued"
    LEASED = "leased"
    RUNNING = "running"
    UPLOADING = "uploading"
    FINISHED = "finished"
    FAILED = "failed"
    CANCELLED = "cancelled"


@dataclass
class Lease:
    task_id: str
    worker_id: str
    fencing_token: int
    expires_at: float
```

数据库条件更新：只有当前 state、token 和未过期租约匹配才能写进度。重新领取时 token 单调增加；旧 Worker 即使恢复，也因 token 落后被拒绝。

3. 控制面与执行面

控制面：Project、Plan、Run、Task、调度、权限、结果索引。执行面：隔离工作目录、拉取固定 commit、安装锁定依赖、运行适配器、上传制品、报告心跳。两者通过小而稳定的 Task 契约连接。

接口适配器输出统一 CaseResult：nodeid、outcome、duration、failure_category、attempt、artifact

refs。Playwright 额外产生 Trace / 视频；pytest API 测试可能产生请求响应摘要。统一核心字段，不抹平专属证据。

4. 隔离与并发

- 每个 Task 独立工作目录、配置和临时数据；
- 数据库使用独立 schema / tenant 或可回滚策略；
- UI 测试每个 Worker 独立账号和 Browser Context；
- 端口、文件、队列名含 worker / run 标识；
- 外部共享资源采用容量令牌，不只靠 xdist 数量。

分片先按历史时长平衡，同时尊重不可拆 fixture 和顺序标记。历史数据缺失时用文件大小 / 用例数估计，运行后更新。

5. 结果与制品

Worker 先上传带内容哈希的制品清单，再提交完成状态；后台对账发现状态 / 制品不一致。CaseResult 区分产品断言、测试 setup、基础设施、超时、取消和跳过，避免把平台故障算成产品缺陷。

重跑创建新的 Attempt，首次失败永不覆盖。平台展示"首跑失败、重跑通过"的 Flaky 状态，并按失败签名聚合。隔离用例有负责人和期限，质量门禁明确是否放行。

6. 可观测与安全

Run 页面从 run_id 关联调度日志、Worker 日志、CaseResult 和 Artifact。监控队列等待、任务耗时、Worker 利用率、租约过期、上传失败和 Flaky 率。仓库凭证短期化，日志 / Trace 脱敏，Worker 禁止不受控出站和宿主机权限。

> 深挖层：Exactly-once 是业务效果，不是消息魔法
> 队列可能至少一次投递。Task Handler 通过唯一键、状态条件、fencing token 和幂等制品上传，把重复投递收敛为一次有效状态转换。不要声称基础设施天然 exactly-once。

> 边界案例：取消与完成同时发生
> 用户取消时 Worker 可能已完成测试并上传结果。状态机定义优先级：例如完成提交先成功则保留 finished 并记录 late_cancel；取消先成功则 Worker 完成写被条件拒绝。不能用最后写入覆盖。

7. 两分钟项目讲述

"平台把分散脚本统一为可追踪 Run / Task / Result。我的核心工作是状态机、租约调度和执行器契约：Worker 崩溃后任务可重领，fencing token 阻止旧 Worker 回写；每次 Run 固定 commit 与配置，首次失败和每次重跑都保留。pytest 与 Playwright 共用结果模型，但保留各自制品。通过并发故障测试、进程强杀、制品对账和 Flaky 统计验证。"

只保留自己确实实现、测试和能解释的主张。

> 练习

> 1. 写出 Task 每条合法转换的数据库条件。
> 2. 模拟 Worker 在上传后、提交状态前崩溃。
> 3. 定义 assertion、setup、infrastructure 三类失败的门禁语义。

关联阅读：C01-C08 测试体系；B09 后台任务；E03-E05 运行与观测。

F04 技术取舍、失败案例与职责边界

速记复习 10-15 分钟；完整阅读约 13 分钟；深挖约 22 分钟

先闭卷回答

1. “为什么不用微服务”怎样回答才不是立场之争？
2. 如何讲失败而不推责也不自我否定？
3. 团队成果与个人贡献如何区分？
4. 不知道答案时，怎样展示工程判断？

> 速记层
> 取舍从目标与约束出发，列候选、决定、代价和复审条件。失败按信号、影响、判断、动作、恢复和系统改进讲。个人贡献用“我负责 / 我决定 / 我验证”，团队成果用“我们”。不知道时先声明边界，再给验证路径，绝不编造。

1. 决策记录模板

```
Context: 当前规模、风险、团队、期限与必须满足的质量属性
Options: 至少两个真实候选
Decision: 选择及最重要理由
Consequences: 获得什么、牺牲什么、新风险是什么
Revisit: 触发重新评估的量化条件
```

例：MVP 选择模块化单体，因为团队小、事务边界集中、部署简单；代价是进程级扩缩不独立。若报告任务 CPU 占用持续影响 API SLO，拆出异步 Worker，而不是因为“微服务更高级”。

2. 常见取舍轴

决策	关注轴
同步 / 异步	用户等待、任务时长、失败恢复、复杂度
线程 / 进程 / asyncio	CPU / I/O、隔离、共享状态、库兼容
页码 / 游标分页	随机跳页、数据变化、索引、复杂度
缓存 / 直读	延迟、数据新鲜度、失效、容量
单体 / 服务拆分	团队边界、部署、事务、可观测成本
自建 / 第三方	差异化价值、锁定、合规、运维

答案必须绑定当前场景；只罗列优缺点而不做决定，不能体现判断。

3. 失败故事结构

用“观察到什么”而不是“某人写错了”：

1. 情境与影响；

2. 最初信号和当时信息；
3. 自己做的假设与动作；
4. 哪个判断不完整；
5. 如何止损、验证和恢复；
6. 增加了什么系统控制；
7. 后续数据证明是否有效。

可讲测试没覆盖的时区边界、连接池在扩容后耗尽、Mock 位置错误掩盖真实调用等。重点是学习速度和系统改进，不是选最惨事故。

4. 职责边界

如果项目是个人完成，说明需求来源和评审方式；如果团队完成，精确说自己拥有的模块、决策和测试。避免把“参与”包装成主导，也避免把团队成果全说成“我”。

证据问答：

- 你写了哪部分？指向模块和接口；
- 谁决定方案？说明提案、评审与最终责任；
- 如何验证？说自己执行的测试和指标；
- 遇到冲突？说依据和实验，不说“我说服了所有人”。

5. 不知道时的答法

“我没有在生产用过自由线程构建。根据 CPython 文档，它移除了 GIL 这一全局限制，但扩展兼容和共享状态同步仍需验证。我会先确认目标 Python 构建与依赖支持，再用竞态测试和 Profile 比较。”

这比猜 API 更专业：明确已知、未知、风险和验证动作。

> 深挖层：高级工程师的信号是边界清晰
> 知道方案在哪些条件成立、如何失败、如何观测和何时重评，比堆叠框架名更有区分度。每个“最佳实践”都应能说出适用上下文。

> 边界案例：无法量化业务收益
> 不要编数字。可量化工程过程：执行时长、失败恢复、人工步骤、缺陷发现、回归覆盖。若连这些也未测，就诚实描述功能结果并把建立指标列为改进。

6. 面试与练习

问：你会重新设计项目的哪部分？

选一个真实代价，说明当时为什么合理、现在出现了什么新证据、如何渐进迁移和验证。不要把整个项目否定，也不要声称没有可改之处。

> 练习
>
> 1. 为一个架构决定写 Decision Record 和复审条件。

> 2. 准备一个失败故事，删除所有甩锅和模糊主语。
> 3. 练习三个不知道的问题：分别给已知、风险和验证路径。

关联阅读：本书 B08-B09、E01-E07 的取舍、失败与恢复边界。

F05 Python 高频面试：从结论下钻到源码

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 30 分钟

先闭卷回答

1. 一个 60 秒技术答案应有几层？
2. “Python 参数是引用传递”为什么不够准确？
3. GIL 问题怎样避免背诵式回答？
4. 解释 MRO 时如何给出可运行证据？

> 速记层
> 高频题用四句结构：结论、机制、最小例子、边界 / 工程影响。先回答问题，再下钻 CPython；不要用实现细节冒充语言保证。所有核心题都能回到对象模型、名称绑定、协议、异常、导入和并发六条主线。

1. 60 秒回答框架

1. 10 秒：一句准确结论；
2. 20 秒：解释运行机制；
3. 15 秒：给一个最小代码或项目例子；
4. 15 秒：补版本、反例或工程边界。

如果面试官追问源码，再说关键结构 / 调用路径；不应一上来背 C 文件名而没回答语义。

2. 高频题索引

问题	结论锚点	深挖单元	必说边界
`is` 与 `==`	身份 vs 值相等	A04	小整数 / 字符串驻留不可依赖
可变默认参数	默认对象定义时创建	A11	用 None 哨兵
深浅拷贝	外层复制 vs 递归复制	A04	自定义对象与共享资源
闭包晚绑定	查找自由变量而非冻结值	A12	默认参数可冻结当前值
装饰器	Callable 包装与名称重绑定	A13	元数据、Descriptor、async
迭代器 / 生成器	单遍协议与暂停帧	A14	耗尽、关闭、异常注入
MRO / super	C3 线性化与协作调用	A16	super 不是固定父类
Descriptor	属性访问协议	A17	data descriptor 优先级

Context Manager	enter / exit 保证边界清理	A18	不能保证进程强杀清理
Import 缓存	`sys.modules` 与执行一次	A19	循环导入、导入副作用
类型提示	静态契约，默认不运行时强制	A20	Any、Variance、Narrowing
GIL	传统 CPython 执行 Python 字节码的互斥	A21	I/O 释放、C 扩展、free-threaded
asyncio	单线程协作式调度	A22	同步阻塞会卡事件循环
事务隔离	并发现象与可见性规则	B06	应用不变量、死锁重试
索引最左前缀	联合索引按有序 Key 组织	B07	选择性、范围条件、回表
fixture scope	依赖图与生命周期	C03	大 scope 会放大共享状态
patch 位置	在被测代码查找处替换	C04	spec、Fake 与集成测试

3. 示例：参数传递

准确说法是 call by sharing / 对象引用按值传入：形参是新的名称绑定，初始指向同一对象；修改共享可变对象对外可见，重新绑定形参不影响调用者名称。

```
def change(items: list[int]) -> None:
    items.append(3)  # 修改共享对象
    items = [9]      # 只重绑定局部名称


values = [1, 2]
change(values)
assert values == [1, 2, 3]
```

4. 示例：GIL

传统 CPython 构建中，同一解释器通常一次只有一个线程执行 Python 字节码；阻塞 I/O 和部分 C 扩展可释放 GIL。因此线程能提高 I/O 并发，却通常不加速纯 Python CPU 循环。多进程能并行 CPU，但有进程、序列化和数据共享成本。

Python 3.13 起提供实验性 free-threaded 构建，后续版本持续演进；它不是默认思维的简单删除项。仍有对象级同步、竞态、扩展兼容和性能权衡，答案要先确认目标构建。

5. 示例：MRO

```
class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B, C):
    pass
```

```
assert D.__mro__ == (D, B, C, A, object)
```

C3 保持局部父类顺序和单调性。`super()` 从当前类之后沿 MRO 查找，协作多继承要求各层签名兼容并继续调用。

> 深挖层：语言语义与 CPython 实现
> 例如 dict 保持插入顺序已是语言保证；小整数缓存是实现优化，范围可变且不应依赖。回答时用“Python 保证”或“当前 CPython 通常”明确层次。源码用于解释机制，不用于制造所有实现都必须相同的结论。

> 边界案例：背出源码文件却版本不匹配
> CPython 内部结构会变。引用源码时说版本和稳定概念，例如“3.11 的自适应解释器会改写 / 专门化执行路径”；不要承诺某字段名在 3.14 仍相同。

6. 面试与练习

问：list 和 tuple 的区别？

先答可变性与语义；再说可哈希需要元素也可哈希、tuple 可作 dict Key；性能和内存只是实现层辅助，不应说 tuple “绝对更快”。最后举 API 返回固定记录 vs 需要修改集合的选择。

> 练习
>
> 1. 随机抽表中 5 题，每题录 60 秒并核对四层。
> 2. 给每题补一个“语言保证 / CPython 实现”标签。
> 3. 从 CPython 源码定位一条属性访问或 dict 路径，写三句机制摘要。

官方参考：Python Data Model；Execution Model；CPython Source

F06 输出题、调试题与现场编码

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 25 分钟

先闭卷回答

1. 输出题应按什么顺序推演？
2. `finally` 中 return 为什么危险？
3. 调试题怎样避免随机改代码？
4. 现场编码为何先说不变量和复杂度？

> 速记层
> 输出题按“对象创建 - 名称绑定 - 调用 - 修改 / 重绑定 - 清理”逐步跟踪，不凭直觉。调试先最小复现、读错误链、提出假设、加观测、单变量验证。现场编码先确认输入 / 边界，写简单正确解，再谈复杂度、测试和优化。

1. 可变默认参数

```
def collect(value: int, bucket: list[int] = []) -> list[int]:
    bucket.append(value)
    return bucket


assert collect(1) == [1]
assert collect(2) == [1, 2]
```

默认 list 在 `def` 执行时创建一次。修复：参数默认 `None`，函数内新建；若确实想共享缓存，应显式命名共享状态。

2. 闭包晚绑定

```
functions = [lambda: index for index in range(3)]
assert [function() for function in functions] == [2, 2, 2]

frozen = [lambda index=index: index for index in range(3)]
assert [function() for function in frozen] == [0, 1, 2]
```

第一个闭包共享同一自由变量 cell，调用时循环已结束；默认参数在创建 lambda 时计算，冻结当前值。

3. 浅拷贝

```
original = [[1], [2]]
copied = original.copy()
copied[0].append(9)

assert original == [[1, 9], [2]]
assert copied is not original
assert copied[0] is original[0]
```

外层容器不同，内层对象共享。是否需要 deepcopy 取决于所有权；数据库连接、锁等对象即使能复制，也未必有合理语义。

4. finally 覆盖控制流

```
def surprising() -> int:
    try:
        return 1
    finally:
        return 2


assert surprising() == 2
```

finally 在离开 try 前执行，其中 return 会覆盖原返回值，也会压掉异常。清理放 finally，但避免 return / break / continue 改写控制流。

5. 迭代器耗尽与 async 阻塞

```
values = (number * 2 for number in range(3))
assert list(values) == [0, 2, 4]
assert list(values) == []
```

生成器是单遍迭代器。调试“第二次为空”先检查是否被日志、断言或长度计算消费。

在 `async def` 里调用阻塞数据库驱动或 `time.sleep` 会卡事件循环；仅加 async 关键字不会变成非阻塞。使用

异步库、线程卸载或独立 Worker，并测事件循环延迟。

6. 调试流程

面对“偶发 KeyError”：

1. 保存完整 traceback、输入摘要、版本与 request_id；
2. 找到第一个属于自己代码的帧；
3. 写最小复现，控制随机、时间和并发；
4. 假设是缺字段、竞态、缓存旧 Schema 或错误分支；
5. 增加结构化观测并逐个证伪；
6. 修复根因，新增回归和边界测试。

捕获 `Exception` 后返回空 dict 会让症状推迟到更远处，破坏错误上下文。

7. 现场编码框架

先复述需求和输入约束，给两三个例子；选择数据结构并说不变量；写清晰正确代码；手动走正常、空、重复、极值；最后说时间 / 空间复杂度和生产增强（类型、错误、并发、持久化）。

```
def first_duplicate(values: list[int]) -> int | None:
    seen: set[int] = set()
    for value in values:
        if value in seen:
            return value
        seen.add(value)
    return None
```

时间 O(n)，额外空间 O(n)。若输入是流仍可逐项处理，但无限高基数需要容量策略；若只能用常量空间，必须有取值范围或允许修改输入等额外约束。

> 深挖层：输出题不是记答案
> 用名称绑定、属性查找、迭代协议和异常控制流解释，就能迁移到陌生代码。每一步写出“哪个对象、哪个引用、何时求值”，比记住某段输出更可靠。

> 边界案例：题目依赖实现优化
> 若 `is` 的结果依赖整数缓存或字符串驻留，应指出不可依赖，而不是给跨版本绝对答案。面试中先说明语义，再说本机 CPython 可能观察到的实现结果。

8. 面试与练习

问：程序线上报错但本地正常，怎么办？

对齐 commit、Python / 依赖、配置、数据规模、时区和并发；从请求 ID 找完整链路，保存原始异常；构造最小差异实验。不要先猜“环境问题”并重装一切。

> 练习
>
> 1. 不运行代码推演本单元四段输出，再执行验证。
> 2. 为 first_duplicate 写空、无重复、首元素重复和大输入测试。

> 3. 选择一次真实 Bug，用六步调试流程重写记录。

官方参考：Compound Statements；Expressions；pdb

F07 后端场景题与系统设计

速记复习 10-15 分钟；完整阅读约 15 分钟；深挖约 27 分钟

先闭卷回答

1. 系统设计题第一步为什么不是画组件？
2. 如何从吞吐与延迟估算容量？
3. 一致性、幂等和重试的关系是什么？
4. 方案怎样体现安全和可恢复性？

> 速记层
> 按“需求与约束 - 核心 API / 数据 - 容量 - 主路径 - 失败路径 - 观测安全 - 演进”作答。先做单区域可运行主线，再识别瓶颈。任何异步和重试都要说明幂等、状态与对账；任何缓存都要说明 Key、TTL、失效和一致性。

1. 七步设计框架

1. 澄清用户、读写比例、延迟、可用性、数据一致性与合规；
2. 定义核心 API、错误语义和幂等；
3. 建模实体、索引、生命周期和数据保留；
4. 粗估 QPS、对象大小、存储、并发和连接；
5. 画正常路径并确定事务边界；
6. 讲超时、重试、重复、乱序、宕机和降级；
7. 加观测、安全、发布和下一阶段扩展。

没有规模就先问；面试官让自行假设时，明确写出假设再算，不追求假精确。

2. 场景：生成大报告

API：`POST /reports` 接受项目、时间窗和幂等键，验证权限后创建 ReportJob，返回 202 + job_id。Worker 领取任务，分块查询、计算、写临时制品，校验后提交完成。`GET /reports/{id}` 返回状态和受权下载引用。

关键点：

- 唯一键 `(actor_id, idempotency_key)` 与 input_hash；
- 状态 queued / running / completed / failed / cancelled；
- 租约和 fencing token；
- 制品内容哈希与原子引用；
- 查询时间 / 行数上限；
- 取消为协作请求，已完成竞态有定义；

- 失败保留可安全重试分类。

3. 场景：高并发热点读取

先确认是否真的热点、数据新鲜度和回源容量。Cache-Aside 读取：缓存 miss 查数据库并写缓存；写入先提交数据库，再删除 / 更新缓存。热点 Key 加 single-flight 或互斥重建，TTL 加抖动防雪崩；空结果短 TTL 防穿透。

缓存 Key 含租户、资源、权限口径和版本。不能把管理员结果复用给普通用户。删除缓存失败要重试 / 订阅变更，接受短暂不一致时明确窗口。

4. 场景：第三方 API 不稳定

设置连接、读取和总超时；只对可重试错误与幂等操作做有限指数退避 + 抖动；尊重 Retry-After；熔断器防持续施压；并发隔离防一个依赖占满线程池。写操作超时先用外部幂等键 / 查询接口核对。

降级返回缓存旧数据时标注 `stale=true` 和 as_of；不能把旧数据伪装成实时。

5. 容量思考

若峰值 100 请求 / 秒，平均服务时间 0.2 秒，则稳定时约有 20 个在途请求；P95 变为 2 秒时在途量会显著增加。Worker、连接池和队列需在下游容量内设置上限。数据库连接不是越多越快，过多并发会增加上下文切换与锁争用。

存储估算写出记录大小 × 每日数量 × 保留天数 × 索引 / 副本系数。数字不必完美，重点是发现“是否内存可放”“是否需要分区 / 归档”。

6. 安全与观测默认进入设计

入口认证，资源级授权，数据库最小权限，Secret 管理，审计高风险动作；输入大小、频率和出站网络有限制。指标覆盖成功、延迟、队列、池、重试、幂等冲突；Trace 串联 API、队列、Worker、SQL 和对象存储。

> 深挖层：一致性选择要落到用户语义
> 不要只说“最终一致”。说明用户创建报告后立即查询会看到 queued；完成事件与下载引用如何原子关联；缓存最多旧多久；重复请求得到同一 job 还是新 job；对账如何修复中间状态。

> 边界案例：消息已处理但 ACK 丢失
> 队列再次投递同一消息。消费者用业务唯一键 / 状态条件判断已完成，返回成功 ACK；如果副作用在外部系统，使用相同幂等键或查询外部状态。不能靠内存 set 去重。

7. 面试与练习

问：如何设计限流？

先定义保护谁和按什么身份：用户、租户、Token、IP 或 Tool。选择 Token Bucket 允许有限突发，状态存本地或共享层；返回明确错误和 Retry-After。网关限流防粗粒度洪峰，应用再按业务资源限流，两层口径要协调。

> 练习
>
> 1. 用七步框架设计报告生成，控制在 15 分钟。
> 2. 给缓存方案补全 Key、TTL、失效、穿透与降级。
> 3. 设计第三方写请求超时后的对账流程。

关联阅读：B01-B09 后端与数据库；E01-E07 性能、运行、安全和恢复。

F08 模拟面试、评分与 60 单元掌握法

速记复习 10-15 分钟；完整阅读约 12 分钟

先闭卷回答

1. “看懂了”与“能讲、能写、能排错”有什么差别？
2. 如何用 10-15 分钟复习一个深度单元？
3. 模拟面试评分为什么必须记录证据？
4. 何时一个知识点可标记为已掌握？

> 速记层
> 每次先闭卷召回，再读速记，最后用代码 / 边界验证。掌握标准是：30 秒结论、2 分钟机制、可运行最小例、一个失败边界、一个项目映射。错题按“概念、机制、代码、场景、表达”分类，用间隔复习而不是连续重读。

1. 10-15 分钟单元循环

时间	动作	产物
0-2 分钟	闭卷回答四问	暴露记忆缺口
2-5 分钟	阅读速记与主线	修正结论
5-9 分钟	手写 / 运行最小代码	验证机制
9-12 分钟	解释边界与项目应用	迁移到工程
12-15 分钟	写一卡片并排下次复习	形成检索线索

不要一开始逐字重读。检索失败本身是学习信号；读完觉得熟悉不等于能独立生成答案。

2. 五级掌握度

级别	标准
0 未见	无法定义
1 识别	看答案觉得熟悉
2 召回	能说结论和简单例子

- 周三：C 测试 2 单元；
- 周四：D MCP / AI 1 单元 + E 运行 1 单元；
- 周五：F 项目 / 面试 2 单元；
- 周末：一次 45 分钟模拟，回填错题。

每天只需一个 10-15 分钟主单元；第二单元可只做四问。高风险缺口重复出现时打破轮换，立即安排代码证据。

7. 版本维护

每季度核对 Python 稳定版、CPython 文档、Flask / pytest / Playwright、MCP 规范和 SDK 主版本。升级知识条目时记录“旧结论、变化、影响、验证代码”，而不是把旧内容静默覆盖。

> 深挖层：掌握是可观察能力
> 同一概念至少通过四种检索：定义题、输出题、Bug、项目取舍。只会答定义说明记忆与情境绑定过窄；能在陌生故障中识别同一机制，才是迁移。

> 边界案例：题库分数越来越高
> 可能只是记住题序。随机化题目、改变变量、换成真实代码审查，并让他人追问。评估应使用未见变式，否则高分不能证明迁移。

8. 最终自测

从六篇各抽一题，要求：30 秒结论、2 分钟机制、3 分钟代码 / 图、1 分钟边界。任何一段依赖看书就回到对应单元的练习；全部完成后，再做项目级串联。

问：什么时候算准备好面试？

不是 60 单元全到 5 级，而是目标岗位高频能力达到 3-4 级；两个项目证据完整；三轮模拟没有准确性红线；未知问题能给出诚实验证路径；现场编码能稳定交付简单正确解。

> 练习
>
> 1. 今天随机抽一个单元，执行完整 15 分钟循环。
> 2. 建立首份错题记录，并安排 1 / 3 / 7 / 21 天复习。
> 3. 完成一轮 45 分钟模拟并按六项评分。

使用建议：结合附录的 60 单元索引、练习检查点和官方版本入口持续维护。

附录 A：练习答案与自检关键点

正文练习多为开放工程题，本附录给出“答案必须覆盖的检查点”，不是唯一实现。先独立完成，再对照；能解释为什么、能运行验证、能处理反例，才算通过。

A01-A07：运行、对象与基础类型

- A01：应能区分解析、AST、代码对象和执行；重复普通 import 复用 `sys.modules`；`.pyc` 是版本相关缓存；动态执行不处理不可信输入。
- A02：赋值建立绑定而非复制；原地修改影响共享对象，重绑定只改名称；`del` 删除绑定；`id` 不可持久化或跨进程比较。
- A03：参数把实参对象绑定到局部形参；修改可变实参可见，重绑形参不可见；默认参数定义时求值；API 所有权应明确“借用、复制或接管”。
- A04：`is` 比身份，`==` 调用值语义；相等对象作为 Key 时需保持相同哈希；浅拷贝共享嵌套对象；深拷贝对连接、锁等资源没有通用正确语义。
- A05：`bool` 是 `int` 子类但业务不应混用；浮点按二进制近似，金额常用 Decimal；`None` 用 `is`；真值判断可能把 0、空值和缺失混在一起。
- A06：文本是 Unicode 字符串，传输 / 存储是 bytes；编码和解码方向准确；文件 / 网络边界显式编码与错误策略；切字节可能破坏多字节字符。
- A07：切片创建新序列但元素仍共享；负步长的 start / stop 语义要用 `slice.indices()` 验证；tuple 可哈希还要求元素可哈希；高频队头操作用 deque。

A08-A14：容器、函数与迭代

- A08：dict / set 依赖哈希与相等；Key 生命周期内哈希语义稳定；冲突由相等比较解决；插入顺序是语言保证，但集合顺序不作为业务协议。
- A09：先按访问模式选容器；排序稳定，Key 通常每元素计算一次；复杂度要结合常数、数据规模和内存；推导式不适合复杂副作用。
- A10：`for` 驱动迭代协议；循环 `else` 在未 break 时执行；`return` / 异常仍执行 finally；分支应覆盖非法状态而不是静默落入默认。
- A11：参数绑定顺序能处理 positional-only、keyword-only、`*args`、`**kwargs`；可变默认值用 None 哨兵；`inspect.signature().bind()` 可验证调用绑定。
- A12：LEGB 查找、赋值使名称成为局部；闭包捕获 cell，默认晚绑定；`nonlocal` 修改最近外层函数绑定；全局可变状态妨碍并发与测试。
- A13：装饰器在定义时应用并重新绑定名称；使用 `functools.wraps`；同步 / 异步包装要保持调用协议；方法装饰器还涉及 Descriptor 绑定。
- A14：Iterable 提供迭代器，Iterator 维护单遍状态；生成器保存 Frame 并在 yield 暂停；消费后耗尽；`close()` / `GeneratorExit` 可清理但强杀不保证。

A15-A22：对象系统、类型与并发

- A15：属性查找需说清实例、类和 Descriptor 优先级；函数作为类属性通过描述符绑定方法；类属性可被实例遮蔽；`__getattribute__` 错误实现会递归。

- A16：C3 MRO 保持局部顺序与单调性；`super()` 沿 MRO 继续而非固定父类；协作多继承要求兼容签名；业务复用默认先考虑组合。
- A17：dataclass 减少样板但不自动保证不变量；ABC 做运行时抽象，Protocol 做结构化静态契约；data descriptor 优先于实例字典；元类在类创建阶段介入，应谨慎使用。
- A18：只捕获能处理的异常并保留链；context manager 的 `__exit__` 返回真会压制异常；文本文件显式编码；原子替换仍需考虑 fsync、权限和跨文件系统。
- A19：绝对 / 相对导入基于包上下文；循环导入常因顶层执行顺序；应用工厂减少导入副作用；`python -m package.module` 与直接运行文件的包语义不同。
- A20：注解默认不强制运行时类型；用 Union / Protocol / TypeVar 表达真实关系，少用 Any；类型收窄必须由可验证条件支持；外部输入仍需运行时校验。
- A21：传统 CPython GIL 限制同解释器 Python 字节码并行；I/O 线程仍有价值；CPU 用进程要计序列化；free-threaded 构建仍需锁、依赖兼容和基准。
- A22：Coroutine 只有被 await / 调度才执行；同步阻塞会卡循环；取消在 await 点传播且清理要再抛；TaskGroup 提供结构化并发，超时与资源关闭需要同一预算。

B01-B09：后端与数据库

- B01：HTTP 方法、状态码、Header、Body 分层；幂等描述重复请求的业务效果；客户端断开不证明服务端未执行；代理会改变真实 Scheme / Host / IP 的获取方式。
- B02：应用上下文和请求上下文生命周期不同；`g` 是请求级，不是全局缓存；应用工厂便于配置与测试；流式响应会延长资源生命周期。
- B03：边界校验结构、类型、范围、组合和业务存在性；错误有稳定 code、message、details、request_id；外部错误不泄露堆栈；序列化字段与权限绑定。
- B04：认证识别主体，授权检查对象与动作；密码用专用慢哈希；Session / Token 撤销和轮换语义明确；CSRF、XSS、暴力尝试和对象越权分别控制。
- B05：连接池有限且获取有超时；事务完成前连接不能随意归还；参数绑定处理值，动态标识符需白名单；游标和连接在异常路径也关闭。
- B06：隔离级别限制可见性，但业务不变量可能需要锁或唯一约束；事务短小；死锁是可预期并发结果，可有限重试整个事务；外部 API 不放进长数据库事务。
- B07：联合索引顺序匹配过滤 / 排序；范围条件影响后续利用；覆盖索引减少回表但增加写成本；EXPLAIN 要结合真实数据和实际耗时。
- B08：Cache-Aside 的 Key 含租户与版本；游标分页基于稳定唯一排序；写操作幂等键持久化输入哈希；重试只用于可重试且幂等的操作；限流返回明确恢复语义。
- B09：依赖向领域层内聚；后台任务有持久状态、租约、心跳、幂等和对账；优雅退出不替代恢复机制；健康与观测覆盖队列、池和失败分类。

C01-C08：pytest 与自动化

- C01：单元、集成、契约、端到端按风险分层；TDD 是红 - 绿 - 重构；测试外部行为而非重复实现；关键权限和状态转换优先。
- C02：收集阶段避免导入副作用；pytest assertion rewriting 提供差异；参数 ID 可读且稳定； xfail 要有原因和严格策略，不能掩盖未知失败。
- C03：fixture 是依赖图；yield 后清理即使测试失败也执行，但进程强杀不保证；scope 越大共享越多；

数据工厂比巨大固定样本灵活。

- C04：patch 被测模块查找名称的位置；autospec 限制接口；时间、随机、环境变量通过依赖或 monkeypatch 控制；核心领域优先 Fake，关键边界保留集成测试。
- C05：Flask test client 不启动真实网络；数据库测试用迁移后的真实引擎并隔离；事务回滚不覆盖提交 / 锁场景；外部 API 用契约和少量沙箱测试。
- C06：异步测试使用同一事件循环策略；xdist Worker 独立端口、库和账号；覆盖率不代表断言质量；Flaky 按根因治理而不是无限重跑。
- C07：优先 role、label、test id 等稳定 Locator；自动等待只保证可操作性；断言最终业务状态；虚拟列表、最终一致和多账号需要专门设计；失败保留 Trace 与请求 ID。
- C08：Run 固定 commit、依赖和配置；Task 状态机有租约和 fencing token；结果区分产品 / 测试 / 基础设施；制品上传可重试并对账；Flaky 首次失败永不覆盖。

D01-D06：MCP 与 AI 应用

- D01：Host 管模型与用户控制，Client 对接 Server，Server 强制业务权限；Tool / Resource / Prompt 控制语义不同；新协议逐请求自包含，旧协议兼容仍可能有会话；传输成功不等于业务成功。
- D02：官方 SDK 2.x 使用 `MCPServer` / `Client`；独立 `fastmcp` 是另一框架；类型标注生成结构契约但不替代领域校验；Handler 保持薄，生命周期创建和关闭池。
- D03：名称具体、Schema 消除非法状态、结果结构化且有界；读写分开；高风险动作预览 - 确认 - 执行；幂等键绑定调用者、工具和输入哈希。
- D04：自然语言先映射有限 DSL；值参数绑定，标识符 / 表达式白名单；权限条件来自可信身份；只读账号仍需超时、行数、扫描与并发限制；结果带口径版本和 as_of。
- D05：模型做有限选择，代码执行状态机、权限和副作用；错误分类决定重试；Run 有步骤 / 时间 / 成本预算；离线评测覆盖正常、边界、对抗和历史事故；确认绑定精确输入。
- D06：领域单测、进程内契约、真实传输、Host 兼容和安全测试分层；不做 Token passthrough；防 SSRF、路径逃逸、资源耗尽；新旧协议水平扩展的会话 / 密钥 / 通知边界不同。

E01-E07：性能、部署与安全

- E01：问题陈述含负载、分位数和资源；用 Trace 分层、Profile 定热点；优先减少工作与 I/O；优化同时验证吞吐、错误和正确性；微基准保留环境与分布。
- E02：区分 Python heap、RSS、原生内存、缓存和 FD；tracemalloc 多轮快照；找到强引用链；所有池 / 队列有上限；RSS 不回落不自动等于泄漏。
- E03：生产 WSGI / 进程管理器而非开发服务器；SIGTERM 标记不就绪、drain、保存状态、关闭资源；liveness / readiness / startup 分工；总连接数按副本和 Worker 相乘。
- E04：元数据、锁文件、制品职责不同；CI 一次构建并逐级提升；配置启动时校验且 Secret 不入镜像；迁移采用 expand / migrate / contract；回滚要考虑数据兼容。
- E05：日志解释事件、指标看总体、Trace 串路径；标签低基数；trace_id、request_id、job_id 分工；SLO 围绕用户结果；健康绿不等于关键业务可用。
- E06：对象级授权、参数化 SQL、无 Shell 命令、根路径限制、SSRF 多层校验、不可信数据禁 pickle；最小权限贯穿调用链；Prompt 不能替代服务端权限。
- E07：先影响与止损，保留证据后做可证伪假设；恢复动作可回滚且有预期指标；对副作用未知状态做对账；复盘修触发、放大和检测整条失败链。

F01-F08：项目与面试

- F01：每个主张包含问题、约束、决策、实现、证据、边界；指标说明环境和样本；仓库可运行、测试可复查、制品可追踪；职责使用准确主语。
- F02：风险结果绑定数据时间、口径、规则、代码和证据；Calculator 纯函数，Run 持久状态，Publisher 幂等；MCP 只暴露领域能力；迟到数据和历史重算有政策。
- F03：平台控制面 / 执行面分离；Run 不可变，Task 租约与 token；执行器统一核心结果并保留专属制品；重复消息通过状态条件收敛；取消竞态有定义。
- F04：取舍从约束出发，写候选、决定、代价和复审条件；失败故事给影响、证据、动作、学习；不知道时明确已知 / 未知 / 验证，不编造。
- F05：答案按结论、机制、例子、边界四层；区分语言保证与 CPython 实现；高频题回到对象、绑定、协议、异常、导入和并发主线。
- F06：逐步跟踪求值和绑定；调试从原始错误和最小复现开始；现场编码先澄清边界、写简单正确解、测试，再谈复杂度与优化。
- F07：系统设计七步完整；所有异步说明状态、幂等和对账；所有缓存说明 Key、TTL 与失效；容量估算暴露瓶颈；安全与观测默认纳入。
- F08：先召回后阅读；掌握要能讲、写、排错和迁移；六项评分留可观察证据；错题按 1 / 3 / 7 / 21 天复习，并用未见变式防止题序记忆。

附录 B：源码阅读地图与官方资料

源码阅读方法

1. 先写最小可运行现象和语言层结论；
2. 固定 Python / 框架版本，不用 `main` 分支解释旧环境；
3. 从公开 API 进入，沿一条调用路径追踪；
4. 记录关键对象、状态变化和异常路径，不逐行翻译；
5. 用测试或 `dis` / Profile 验证推断；
6. 最后写清哪些是语言保证、哪些是当前实现。

源码会重构，文件路径是导航线索，不是稳定 API。扩展模块只应依赖官方公开 C API / Stable ABI 约定，不应因读过内部结构就直接访问私有字段。

CPython 路线

主题	常用入口	对应单元
解析与编译	`Parser/`、`Python/compile.c`	A01、A19
字节码执行	`Python/bytecodes.c`、`Python/ceval.c`	A01、A10、A11
基础对象	`Include/object.h`、`Objects/object.c`	A02-A05
list / tuple	`Objects/listobject.c`、`Objects/tupleobject.c`	A07、A09
dict / set	`Objects/dictobject.c`、`Objects/setobject.c`	A08
函数与闭包	`Objects/funcobject.c`、Frame / Cell 相关实现	A11-A13
生成器 / 协程	`Objects/genobject.c`、`Lib/asyncio/`	A14、A22
类型与属性	`Objects/typeobject.c`、Descriptor 相关对象	A15-A17
异常	`Objects/exceptions.c`、求值器异常路径	A18
线程与 GIL	`Python/ceval_gil.c` 等版本对应实现	A21
GC 与内存	`Modules/gcmodule.c`、`Objects/obmalloc.c`	E02

推荐先在目标 Tag 下搜索公开类型 / 函数名，再观察调用者。目录与函数名在 3.11-3.14 之间可能迁移。

框架与工具路线

主题	阅读入口	先回答的问题
Flask	`pallets/flask` 的 app、ctx、testing	请求 / 应用上下文何时创建与弹出？
Werkzeug	request、response、routing、middleware	WSGI environ 如何变成请求对象？

pytest	`_pytest/fixtures.py`、runner、assertion rewrite	fixture 如何解析、断言如何改写、阶段如何报告？
pytest-xdist	scheduler、remote Worker 协议	收集为何必须一致、分片如何下发？
Playwright Python	同步 / 异步 API 包装、pytest plugin	Locator 与自动等待在哪层实现？
MCP Python SDK	`MCPServer`、`Client`、协议类型与传输	Schema 如何生成、版本如何选择、结果如何编码？
FastMCP	Server、Client、middleware、dependencies	依赖如何注入、Context 生命周期是什么？

官方核对入口

Python 与 CPython

- Python 3 Documentation
- Data Model
- Execution Model
- Standard Library
- Free-threaded Python HOWTO
- CPython Source
- Python Enhancement Proposals

后端与数据库

- Flask Documentation
- Flask Application Lifecycle
- Flask Request Context
- Werkzeug Documentation
- MySQL 8.4 Reference Manual
- InnoDB Isolation Levels
- InnoDB Locking
- EXPLAIN

测试

- pytest Documentation
- pytest Fixtures
- Monkeypatch
- pytest-xdist
- Playwright Python
- Playwright Auto-waiting

- Playwright Trace Viewer

MCP 与 AI 应用

- MCP 2026-07-28 Specification
- MCP Key Changes
- MCP Security Best Practices
- Official MCP Python SDK
- MCP Python SDK Docs
- FastMCP Docs

运行、安全与交付

- Python Packaging User Guide
- Docker Multi-stage Builds
- OpenTelemetry Python
- W3C Trace Context
- OWASP Cheat Sheet Series
- Google SRE Books

季度版本维护清单

- 记录本机与生产 `python --version`、构建类型和支持周期；
- 查看 Python “What’s New”、弃用与移除列表；
- 核对 Flask、pytest、Playwright、MySQL 驱动的主版本迁移说明；
- 核对 MCP 规范日期、目标 Host 功能与 `mcp` / `fastmcp` 包版本；
- 运行本书所有可执行代码、项目契约测试和兼容矩阵；
- 对变化写“旧结论 - 新结论 - 影响 - 验证”，保留历史版本。

附录 C：12 组十五分钟抽题卡

使用方式：任选一组，前 8 分钟闭卷回答，后 5 分钟写 / 运行一个最小例，最后 2 分钟说项目应用。答不完整就回到括号中的单元。

卡 01：对象与绑定

1. `b = a`、浅拷贝、深拷贝分别共享什么？（A02、A04）
2. 为什么 `is` 不能比较普通值？（A04）
3. 可变参数在函数内修改与重绑有什么差异？（A03）
4. 写出一个因共享嵌套对象导致的 Bug，并给所有权方案。（A04）

卡 02：容器与复杂度

1. dict Key 为什么必须保持哈希语义稳定？（A08）
2. list、set、deque 分别适合什么访问模式？（A07-A09）
3. 稳定排序如何实现多级排序？（A09）
4. 一个 O(n) 方案何时可能输给 O(n log n)？（A09、E01）

卡 03：函数、闭包与装饰器

1. 默认参数何时求值？（A11）
2. 闭包晚绑定的机制与两个修复是什么？（A12）
3. `functools.wraps` 保留什么，不能保证什么？（A13）
4. 怎样写同时支持同步 / 异步的装饰器边界？（A13、A22）

卡 04：类与协议

1. 属性查找和 data descriptor 优先级是什么？（A15、A17）
2. `super()` 为什么不是调用固定父类？（A16）
3. ABC 与 Protocol 如何选？（A17）
4. dataclass frozen 是否等于深度不可变？（A17）

卡 05：并发与异步

1. 线程、进程、asyncio 的默认选型依据是什么？（A21-A22）
2. 传统 GIL 限制什么，不限制什么？（A21）
3. asyncio 取消如何传播和清理？（A22）
4. 设计一个有背压、超时和优雅退出的并发流程。（A22、B09、E03）

卡 06：Flask 与 API

1. 请求上下文和应用上下文有何不同？（B02）
2. 400、401、403、404、409、422、429 怎样区分？（B01-B04）

3. 鉴权为何必须做对象级检查？（B04、E06）
4. 超时后写请求状态未知怎样处理？（B08）

卡 07：MySQL 与可靠数据

1. 隔离级别不能自动保护哪些业务不变量？（B06）
2. 联合索引顺序如何由查询决定？（B07）
3. 死锁重试需要满足什么条件？（B06、B08）
4. 设计游标分页的稳定排序和 next_cursor。（B08）

卡 08：pytest 与自动化

1. fixture 作用域为何会影响隔离？（C03）
2. patch 为什么在查找处？（C04）
3. xdist 下如何隔离数据库、端口和账号？（C06）
4. Playwright 自动等待不能解决哪些 Flaky？（C07）

卡 09：MCP 接口

1. Tool、Resource、Prompt 如何选择？（D01）
2. 官方 SDK 2.x 与 FastMCP 怎样区分？（D02）
3. 写 Tool 如何实现预览、确认和幂等？（D03）
4. 新旧 MCP 协议的会话与扩展边界是什么？（D01、D06）

卡 10：数据 Agent

1. 为什么不暴露 `execute_sql(sql)`？（D04）
2. 如何让模型生成受限 DSL 而非任意 SQL？（D04）
3. Agent 状态机、预算和错误分类如何设计？（D05）
4. 如何评测事实正确、工具选择和安全遵循？（D05）

卡 11：性能、安全与故障

1. P95 下降为什么可能是假优化？（E01）
2. RSS 增长和 Python 对象泄漏怎样区分？（E02）
3. SSRF 校验为什么必须覆盖 DNS 与重定向？（E06）
4. 事故前 15 分钟怎样止损、采证和验证？（E07）

卡 12：项目与面试

1. 用六段证据链介绍一个项目。（F01）
2. 讲风险工具的版本、证据和发布幂等。（F02）
3. 讲测试平台的租约、token 和结果模型。（F03）
4. 回答一个不知道的问题：已知、未知、风险、验证。（F04）

一页复习记录模板

日期	单元 / 卡片	闭卷得分 0-4	最大缺口	代码 / 证据	下次复习

完成标准

当你能随机完成 12 组中的任意 3 组，并把答案连接到两个项目、一个失败案例和一个可运行实验时，这份手册就不再只是“看过的知识”，而是可调用的 Python 能力系统。