

# Kubectl apply sync問題



# 任務

讓我弄清楚

##為什麼kubectl apply後，造成not sync

##為什麼kubectl edit後，可以解決sync

##到底能不能移除 metadata 下的 creationTimestamp、resourceVersion、uid、generation 字段

# 任務說明

1.將我給你的問題描述好好利用

2.說明並讓我理解

# 數據庫

## 問題描述

用kubectl edit移除這些內容後，原本argocd ui顯示沒有sync，edit完就變成sync

"creationTimestamp""2025-03-

21T06:20:07Z""generation":

6," labels": ("app.kubernetes.io/

instance"."ifond-vmvminsert-pro

d-fab1 8edge-fz]")," name""f1 8-v

mvminsert-virtual-svr-80",nam

espace""ifond", resourceVersio n""1

90657607""uid""94445b8-8926-4baf-a6f7-

5a4767b06ce7"

creationTimestamp:

"2025-03-21 T06:20:07Z"

generation: 7



已思考 38s

🌐 Searching Argo CD sync issues and Kubernetes metadata handling

github.com

argo-cd.readthedocs.io

kubernetes.io



Powered by [ChatGPT Exporter](#)

👤 其他 6 個

🌐 Searching kubectl edit update strategy documentation

🔍 stackoverflow.com

🌐 kubernetes.io

🖥️ komodor.com

👤 其他 7 個

✅ 完成

## 重點總覽（先答結論）

- 不能把 ``creationTimestamp``、``resourceVersion``、``uid``、``generation`` 等 **server-managed** 欄位放在 Git 的宣告檔（或用來做 ``kubectl apply`` 的檔案）——這些欄位由 API server 產生／管理，不應該當作「期望狀態」去維護（也會造成衝突或比較差異）。  
[Stack Overflow](#) [GitHub](#)
- 你看到的「``kubectl apply`` 後 Argo CD 顯示 not sync，但用 ``kubectl edit`` 編輯後變成 synced」常見原因：Git／apply 的檔案或 last-applied-annotation 含有不該存在的 server-managed metadata，或 field ownership (``managedFields``) / apply 策略不同，造成 Argo CD 的 diff 演算覺得 live 與 repo 不同。用 ``kubectl edit`` 會改變 live 的 payload / managedFields / last-applied 註記，間接消除那個差異，因此 UI 變回 sync。  
[argo-cd.readthedocs.io](#) [+1](#)

## 詳細解釋 — 為什麼會出現你描述的行為

### 1) Kubernetes metadata 是誰在管？為什麼不能放在 Git

``metadata.uid``、``metadata.creationTimestamp``、``metadata.resourceVersion``、``metadata.generation`` 等欄位都是 Kubernetes API server **動態產生或維護** 的值，用來表示物件身分、版本與 server 的內部狀態（例如 ``resourceVersion`` 用於樂觀鎖、``generation`` 由 controller 更新以表示 spec 變更）。這些值不是「使用者應該宣告的期望狀態」，因此把它們寫回到你的宣告檔會造成不必要的 diff、衝突或 apply 失敗。 [Kubernetes](#) [GitHub](#)

### 2) Argo CD 如何判斷「in-sync / out-of-sync」

Argo CD 會把 **Git (desired)** 與 **cluster (live)** 進行 diff。預設策略有三種 (legacy 3-way、structured-merge、server-side diff)。Argo CD 也支援忽略特定路徑或由特定 field-manager 所擁有的欄位 (``ignoreDifferences`` / ``managedFieldsManagers``)。但如果 Git 中包含 server-managed 欄位，或不同工具 / manager 改變了同一欄位的擁有權，diff 會顯示為「不同」，導致 out-of-sync。你可以用 ArgoCD 的 diff customization 去忽略某些路徑，但最佳做法是不要把 server 欄位放在 Git。 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io) [Discuss Kubernetes](#)

### 3) ``kubectl apply`` 與 ``kubectl edit`` 的差別 (導致行為不同的原因)

- ``kubectl apply`` (傳統 client-side apply 或 server-side apply 取決於你呼叫方式) 會根據你提供的檔案做三向合併 (client-side) 或由 API server 處理欄位擁有權 (server-side apply)；如果你的檔案把 server 管理的欄位一併包含，會造成不必要的差異或衝突 (或者 API server 忽略/拒絕) — 並且 ``apply`` 依賴 ``last-applied-configuration`` 來做 3-way diff。 [Kubernetes](#) [FAUN.dev — Developer Community](#)
- ``kubectl edit`` 的工作流程是：先 ``GET`` live 物件、在編輯器編輯、然後送回 (用 API 的該版本進行更新)。實務上 ``kubectl edit`` 常常會把「你編輯後的 live 物件」重新寫回，並且不會把那些 server-only 欄位放到你儲存並送出的「期望檔案」中 (使用者通常會刪掉 server 欄位或編輯時沒包含它們)，從而更改了 live 的 **managedFields / last-applied** 註記或直接消除了差異，使 Argo CD 的比較結果變成 synced。簡單講：``apply`` 在用檔案去驅動狀態，``edit`` 是直接改 live，兩者在 field ownership 與 last-applied 註記上會有不同效果。 [Kubernetes](#) [Stack Overflow](#)

## 實務建議 (怎麼修、怎麼避免再次發生)

#### 1. 不要把 server-managed 欄位提交到 Git。

在把 ``kubectl get -o yaml`` 的內容放回 repo 前，請移除：

- ``metadata.creationTimestamp``
- ``metadata.resourceVersion``



- ``.metadata.uid``
- ``.metadata.generation``
- ``.metadata.managedFields``（也常一併去掉）

範例用 ``jq`` / ``yq`` 清理：

```
# 用 jq (JSON)
kubectl get deploy my-deploy -n myns -o json \
  | jq 'del(.metadata.creationTimestamp, .metadata.resourceVersion,
.metadata.uid, .metadata.generation, .metadata.managedFields)' \
  > my-deploy.cleaned.json

# 或用 yq (yaml)
kubectl get deploy my-deploy -n myns -o yaml \
  | yq 'del(.metadata.creationTimestamp, .metadata.resourceVersion,
.metadata.uid, .metadata.generation, .metadata.managedFields)' \
  > my-deploy.cleaned.yaml
```

這樣 commit 到 Git 後 Argo CD 比較時就不會被 server-only 欄位干擾。 [Stack Overflow](#)

2. 如果某些欄位確實會被 **cluster** 或其他 **controller**（例如 **CSI**、**external controllers**）動態修改，考慮用 **Argo CD** 的 ``.ignoreDifferences`` / ``.resource.customizations`` 去忽略那些路徑（但這是其次的解法，最好是讓 Git 為「乾淨的宣告檔」）。 [argo-cd.readthedocs.io](#)

3. 如果出現 **out-of-sync**，可以做的操作：

- 用 ``argocd app diff <app>`` / Argo UI 看哪個欄位不同。 [argo-cd.readthedocs.io](#)
- 如果差異只來自 server-managed 欄位，清理 Git 檔案再 push，然後 ``argocd app refresh <app>`` 或在 UI 點 Refresh。
- 若是 field ownership (managedFields) 混亂，可能需要用 ``kubectl edit`` 或重新 apply 一次乾淨的 manifest（或在某些情況下 delete + recreate 由 Argo CD 管理）來重置 managedFields。注意 delete-recreate 有風險（中斷服務），請評估。  
[argo-cd.readthedocs.io](#)

4. 避免把 ``kubectl get -o yaml`` 的輸出直接丟回 **Git**（因為它包含很多 server 填入欄位）。若真的要採集現況為基底，一定要使用上面的 ``jq`` / ``yq`` 刪除步驟，或使用模板化的 manifest

來源 (Helm / Kustomize / kpt 等) 來維護。 [Stack Overflow](#)

## 小範例：你遇到的情境（對應你貼的片段）

你貼出的片段像是把 live 的 object (含 `"creationTimestamp": "2025-03-21T06:20:07Z"`, `"resourceVersion": "190657607"`, `"uid": "94445b8-..."`, `generation: 7`) 放到某個地方 (可能是 Git 或是用來 `kubectl apply` 的檔案)。

- 如果那個檔案被 Argo CD 當作 desired state 的來源, Argo CD 做 diff 時會看到 repo 裡那組 metadata 與 live cluster 的欄位不一致 → out-of-sync。 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)
- 你直接用 `kubectl edit` 把那些欄位移掉或編輯後, live 的 managedFields / last-applied 可能被更新、或 Argo CD 在下一次計算 diff 時找不到差異, 故顯示為 synced (但本質上你只是把 Git 與 live 的「比較基準」調整到一致)。 [Kubernetes](#) [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)

## 結論（一步到位的修法）

1. 從 Git 或任何自動化 pipeline 移除 server-managed 欄位 (用 `jq`/`yq` 或確保生成流程不輸出它們)。`
2. Commit 乾淨 manifest, 讓 Argo CD 以「純宣告檔」比對與同步。
3. 若某些差異是必然 (controller 會改某些欄位), 配置 Argo CD 的 `ignoreDifferences` 或 `managedFieldsManagers` 忽略那些差異 (但優先還是把 Git 淨化)。 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io) +1

如果你要, 我可以直接：

- 幫你把你貼的那段 YAML 轉成「可 commit 的乾淨版本」(我會示範 ``jq`/`yq`` 刪欄位後的結果)，或
- 寫一個 CI / git-hook 小命令 (``jq`` 或 ``yq``) 示例，用來在提交前自動清 metadata。

要哪一項我就直接幫你做 (不用再問細節)，或把你貼過來的完整 YAML 丟給我，我馬上幫你清掉那些欄位並回傳乾淨的 YAML。



#### # 任務

讓我弄清楚

##為什麼kubectl apply後，造成not sync

##為什麼kubectl edit後，可以解決sync

##到底能不能移除 metadata 下的 creationTimestamp、resourceVersion、uid、generation 字段

#### # 任務說明

1.將我給你的問題描述好好利用

2.說明並讓我理解

#### # 數據庫

##### ## 問題描述

用kubectl edit移除這些內容後，原本argocd ui顯示沒有sync，edit完就變成sync

"creationTimestamp""2025-03-

21T06:20:07Z""generation":

6," labels": ("app.kubernetes.io/

instance"."ifond-vmvminsert-pro

d-fab1 8edge-fz]")," name""f1 8-v

mvminsert-virtual-svr-80",nam

espace""ifond", resourceVersio n""1

90657607""uid""94445b8-8926-4baf-a6f7-

5a4767b06ce7"

creationTimestamp:

"2025-03-21 T06:20:07Z"

generation: 7



為了幫你深入釐清 `kubectl apply` 導致 Argo CD out-of-sync、而 `kubectl edit` 又能讓它變成 synced 的具體原因，我想請你確認幾個細節：

1. 你說的是 Argo CD UI 的 sync 狀態對吧？不是 `kubectl` 的 output？
2. 你用 `kubectl apply` 時，是從哪個來源 apply？是 Git repo 中的檔案、還是你從 `kubectl get` 出來另存的 YAML？
3. 那些 metadata 欄位（如 `creationTimestamp`、`uid` 等）原本是在 Git 裡就存在，還是從 live 物件複製過來加進去的？

有了這些細節，我就可以更準確地說明背後的行為差異和最佳做法。



1. 是 Argo CD UI 的 sync 狀態
2. 我從 `kubectl get` 出來另存的 YAML？
3. 從 live 物件複製過來加進去的？



了解了，我將深入說明為什麼你用 `kubectl apply` 一個從 live 物件複製來的 YAML（包含 metadata 欄位）後會造成 Argo CD 顯示 out-of-sync，而改用 `kubectl edit` 卻能讓它變回 synced，以及這些 metadata 欄位到底應不應該被移除。

我會針對 Argo CD 的 diff 機制、Kubernetes metadata 欄位的性質、`kubectl` 的行為差異，並提供清晰建議與範例說明。稍後我會回覆你完整分析報告。

🔗 Argo CD Out-of-Sync 原因與 kubectl apply/edit 差異分析



# Argo CD 同步原理

Argo CD 以 Git 仓库中的声明性资源清单作为「期望状态」，定期对照 Kubernetes 集群中的「实际状态」进行比对。如果二者有任何差异，Argo CD UI 就会显示 **OutOfSync**（不同步），否则显示 **InSync**。文档指出，如果清单（manifest）中包含了实际状态中不存在或多余的字段，这些字段在查询集群资源时会被丢弃，从而检测到差异并报出 OutOfSync [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)。换言之，Argo CD 检测差异的逻辑是针对【Git 清单 ↔ 集群实际对象】进行的三路合并比较，一旦发现 Git 清单与集群对象有不同，就会标记为 OutOfSync。

## kubectl apply 导致 OutOfSync 的原因

- **手动 `kubectl apply` 与 GitOps 冲突**：在 Argo CD 管理下的资源，如果你在集群上使用 `kubectl apply` 手动更新，而未同步回 Git，会改变实际对象状态但不更新 Git 清单，Argo CD 会认为这是“漂移”。Argo CD 默认使用 **kubectl apply** 的三方合并模式，同步时会在集群对象的 `kubectl.kubernetes.io/last-applied-configuration` 注释中记录上次应用的状态 [stackoverflow.com](https://stackoverflow.com)。如果你手动应用的 YAML 与 Git 中的声明不一致，Argo 会发现不同而报告 OutOfSync。
- **元数据字段冲突**：从集群导出的 YAML 通常包含大量元数据字段（如 `creationTimestamp`、`resourceVersion`、`uid`、`generation` 等）。这些字段都是 Kubernetes 自动生成或内部使用的，不属于用户声明的规范内容。如果将包含这些字段的 YAML 再次用 `kubectl apply`，会触发合并冲突。例如，如某回答指出，`kubectl apply` 会基于本地文件、集群实时状态以及 `last-applied-configuration` 注释执行三向合并 [stackoverflow.com](https://stackoverflow.com)。如果 `last-applied-configuration` 中意外含有 `resourceVersion` 等字段，而本地文件没有，对比时就会尝试将该字段从集群对象中删除，导致冲突报错 [stackoverflow.com](https://stackoverflow.com)。这种冲突也会



让 Argo CD 检测到“集群对象与 Git 清单不一致”，从而显示 OutOfSync。实际上，官方工具给出的错误信息类似：

```
Operation cannot be fulfilled on <kind>/<name>: the object has been modified; please apply your changes to the latest version
```

这表明包含在 YAML 里的集群元数据（如 `resourceVersion`）与当前对象不符，必须删除它们再重试 [stackoverflow.com](https://stackoverflow.com)。

- **示例流程：**当执行 `kubectl apply -f file.yaml` 时，如果 `file.yaml` 是通过 `kubectl get -o yaml` 导出的原始对象，其内含有上述元数据字段，那么 `apply` 过程会试图删除这些字段，引发错误或部分忽略操作。即使没有明显错误，最终集群中的资源状态（尤其是这些元数据）也会与 Git 存储的清单不一致，导致 Argo CD 报告 OutOfSync。

## kubectl edit 删除元数据字段后恢复同步的原因

当你在集群上用 `kubectl edit` 手工删除了那些自动生成的元数据字段后，集群对象中的实际状态就与 Git 清单中的描述完全一致了。因为 Git 清单通常**不包含**这些字段（它们是集群管理的结果），所以删除后 Argo CD 再次比较时不会检测到差异，Sync 状态就变成了 InSync。

- **对比校验：**例如，如果 Git 清单只描述了 Pod 的 `spec`，而不包含 `metadata.creationTimestamp`、`metadata.resourceVersion`、`metadata.uid`、`metadata.generation` 等字段，那么集群对象实际上也不“期望”这些字段匹配任何值。移除这些字段后，集群中对象的剩余内容（名称、标签、spec 等）与 Git 清单一致，Argo CD 不再发现多余或缺失字段，于是同步标记正常恢复。
- **Argo CD 差异检测逻辑：**如前所述，Argo CD 对比的是 Git 中的清单与“查询到的集群对象”。如果集群对象在 `metadata` 部分只剩下名称、标签、命名空间等关键字段，而不携带资源版本等可变字段，Argo 的比较过程就不会认为有差异。简而言之，手工移除元数据中的动态字段，使得声明式描述（Git）和实际状态保持一致，从而解决了 OutOfSync。

# 是否可以移除 metadata 下的字段

- **字段性质：**``metadata.creationTimestamp``、``metadata.resourceVersion``、``metadata.uid``、``metadata.generation`` 等字段都是由 Kubernetes 控制平面自动维护的内部信息。它们不是用户在 YAML 清单中指定的内容，而是集群在创建或更新对象时自动产生的。按照最佳实践，这些字段应当从声明式配置中 **省略**。Kubernetes 官方指南也指出，元数据下的某些键并不是必需的，例如可以删除 ``creationTimestamp`` 和 ``selfLink`` [blog.csdn.net](https://blog.csdn.net)。用户只需在 YAML 中保留必要的 ``name``、``namespace``、``labels`` 等，其他由集群管理的字段都不应手动填写。
- **安全移除：**从 YAML 清单中删除这些字段是安全且推荐的做法。正如文档和社区经验所示，这些字段一旦在清单中被删除，集群在创建/更新时会自动填写它们，反之若清单包含它们，则可能引发冲突 [stackoverflow.com](https://stackoverflow.com)。例如一位回答者总结：“删除 ``spec.clusterIP``、``metadata.uid`` 等**集群特有字段**后，再去部署不会受影响；这些字段是由集群自动生成或维护的” [stackoverflow.com](https://stackoverflow.com)。**结论：可以且应该移除这些字段**，让 Git 清单只描述真正的声明内容，由 Kubernetes 控制器来管理内部元数据。
- **Argo CD 对比行为：**在 Argo CD 的对比机制中，``resourceVersion`` 和 ``generation`` 等字段通常不被视为声明性的一部分（它们在 Git 中不存在）。实际上，如果 Git 清单不包含它们，Argo CD 在计算 diff 时并不会认为需要将这些字段“补回”清单。正相反，如果清单里出现了这些字段，Argo CD 会识别为多余字段并报告差异 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)。因此，最简便的做法是，在 Git 中完全不出现这些字段，真正的效果是 Argo CD 不会将它们视为同步目标的一部分。

## 小结

- **不要将集群自动填充的元数据带入 Git 清单。**例如通过 ``kubectl get -o yaml`` 导出的完整对象通常包含 ``creationTimestamp``、``resourceVersion``、``uid``、``generation`` 等字段，这些只在集群当前状态有效。将其复制到 Git 仓库会导致 Argo CD 视其为多余字段，从而触发 OutOfSync。 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io) [blog.csdn.net](https://blog.csdn.net)
- **使用 GitOps 时避免手动 ``kubectl apply`` 修改。**任何绕过 Git 直接在集群上做的变更（包括 ``kubectl apply``、``kubectl edit`` 等）都会造成 Git 与集群状态不一

致。如需变更，应先更新 Git，再让 Argo CD 同步 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io) [stackoverflow.com](https://stackoverflow.com)

。

- **移除元数据字段后恢复同步。**正如经验所示，手动编辑并删除元数据中这类字段后，Argo CD 会检测到状态一致，从而显示为 InSync。这验证了这些字段本身并不是声明性配置的一部分，只要不在 Git 清单中出现即可。特别提醒：始终在 Git 中只保留 ``metadata.name``、``namespace``、``labels`` 等必要项，其他字段由集群自动维护

[blog.csdn.net](https://blog.csdn.net) [stackoverflow.com](https://stackoverflow.com) 。

**参考资料：**Argo CD 官方文档 [argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)、Kubernetes 配置示例 [blog.csdn.net](https://blog.csdn.net)、相关社区讨论 [stackoverflow.com](https://stackoverflow.com) [stackoverflow.com](https://stackoverflow.com) 等。



轉成繁體中文:

Argo CD 同步原理

Argo CD 以 Git 仓库中的声明性资源清单作为「期望状态」，定期对照 Kubernetes 集群中的「实际状态」进行比对。如果二者有任何差异，Argo CD UI 就会显示 OutOfSync（不同步），否则显示 InSync。文档指出，如果清单（manifest）中包含了实际状态中不存在或多余的字段，这些字段在查询集群资源时会被丢弃，从而检测到差异并报出 OutOfSync

[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)

。换言之，Argo CD 检测差异的逻辑是针对【Git 清单 ↔ 集群实际对象】进行的三路合并比较，一旦发现 Git 清单与集群对象有不同，就会标记为 OutOfSync。

kubectl apply 导致 OutOfSync 的原因

手动 kubectl apply 与 GitOps 冲突：在 Argo CD 管理下的资源，如果你在集群上使用 kubectl apply 手动更新，而未同步回 Git，会改变实际对象状态但不更新 Git 清单，Argo CD 会认为这是“漂移”。Argo CD 默认使用 kubectl apply 的三方合并模式，同步时会在集群对象的 `kubectl.kubernetes.io/last-`

applied-configuration 注释中记录上次应用的状态  
stackoverflow.com

。如果你手动应用的 YAML 与 Git 中的声明不一致，Argo 会发现不同而报告 OutOfSync。

元数据字段冲突：从集群导出的 YAML 通常包含大量元数据字段（如 creationTimestamp、resourceVersion、uid、generation 等）。这些字段都是 Kubernetes 自动生成或内部使用的，不属于用户声明的规范内容。如果将包含这些字段的 YAML 再次用 kubectl apply，会触发合并冲突。例如，如某回答指出，kubectl apply 会基于本地文件、集群实时状态以及 last-applied-configuration 注释执行三向合并  
stackoverflow.com

。如果 last-applied-configuration 中意外含有 resourceVersion 等字段，而本地文件没有，对比时就会尝试将该字段从集群对象中删除，导致冲突报错  
stackoverflow.com

。这种冲突也会让 Argo CD 检测到“集群对象与 Git 清单不一致”，从而显示 OutOfSync。实际上，官方工具给出的错误信息类似：

Operation cannot be fulfilled on <kind>/<name>: the object has been modified; please apply your changes to the latest version

这表明包含在 YAML 里的集群元数据（如 resourceVersion）与当前对象不符，必须删除它们再重试  
stackoverflow.com

。

示例流程：当执行 kubectl apply -f file.yaml 时，如果 file.yaml 是通过 kubectl get -o yaml 导出的原始对象，其内含有上述元数据字段，那么 apply 过程会试图删除这些字段，引发错误或部分忽略操作。即使没有明显错误，最终集群中的资源状态（尤其是这些元数据）也会与 Git 存储的清单不一致，导致 Argo CD 报告 OutOfSync。

kubectl edit 删除元数据字段后恢复同步的原因

当你在集群上用 kubectl edit 手工删除了那些自动生成的元数据

字段后，集群对象中的实际状态就与 Git 清单中的描述完全一致了。因为 Git 清单通常不包含这些字段（它们是集群管理的结果），所以删除后 Argo CD 再次比较时不会检测到差异，Sync 状态就变成了 InSync。

对比校验：例如，如果 Git 清单只描述了 Pod 的 spec，而不包含 metadata.creationTimestamp、metadata.resourceVersion、metadata.uid、metadata.generation 等字段，那么集群对象实际上也不“期望”这些字段匹配任何值。移除这些字段后，集群中对象的剩余内容（名称、标签、spec 等）与 Git 清单一致，Argo CD 不再发现多余或缺失字段，于是同步标记正常恢复。

Argo CD 差异检测逻辑：如前所述，Argo CD 对比的是 Git 中的清单与“查询到的集群对象”。如果集群对象在 metadata 部分只剩下名称、标签、命名空间等关键字段，而不携带资源版本等可变字段，Argo 的比较过程就不会认为有差异。简而言之，手工移除元数据中的动态字段，使得声明式描述（Git）和实际状态保持一致，从而解决了 OutOfSync。

是否可以移除 metadata 下的字段

字段性质：metadata.creationTimestamp、metadata.resourceVersion、metadata.uid、metadata.generation 等字段都是由 Kubernetes 控制平面自动维护的内部信息。它们不是用户在 YAML 清单中指定的内容，而是集群在创建或更新对象时自动产生的。按照最佳实践，这些字段应当从声明式配置中省略。Kubernetes 官方指南也指出，元数据下的某些键并不是必需的，例如可以删除 creationTimestamp 和 selfLink  
[blog.csdn.net](https://blog.csdn.net)

。用户只需在 YAML 中保留必要的 name、namespace、labels 等，其他由集群管理的字段都不应手动填写。

安全移除：从 YAML 清单中删除这些字段是安全且推荐的做法。正如文档和社区经验所示，这些字段一旦在清单中被删除，集群在创建/更新时会自动填写它们，反之若清单包含它们，则可能引发冲突  
[stackoverflow.com](https://stackoverflow.com)

。例如一位回答者总结：“删除 spec.clusterIP、.metadata.uid 等集群特有字段后，再去部署不会受影响；这些字段是由集群自

动生成或维护的”

[stackoverflow.com](https://stackoverflow.com)

。结论：可以且应该移除这些字段，让 Git 清单只描述真正的声明内容，由 Kubernetes 控制器来管理内部元数据。

Argo CD 对比行为：在 Argo CD 的对比机制中，resourceVersion 和 generation 等字段通常不被视为声明性的一部分（它们在 Git 中不存在）。实际上，如果 Git 清单不包含它们，Argo CD 在计算 diff 时并不会认为需要将这些字段“补回”清单。正相反，如果清单里出现了这些字段，Argo CD 会识别为多余字段并报告差异

[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)

。因此，最简便的做法是，在 Git 中完全不出现这些字段，真正的效果是 Argo CD 不会将它们视为同步目标的一部分。

## 小结

不要将集群自动填充的元数据带入 Git 清单。例如通过 `kubectl get -o yaml` 导出的完整对象通常包含 `creationTimestamp`、`resourceVersion`、`uid`、`generation` 等字段，这些只在集群当前状态有效。将其复制到 Git 仓库会导致 Argo CD 视其为多余字段，从而触发 `OutOfSync`。

[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)

[blog.csdn.net](https://blog.csdn.net)

使用 GitOps 时避免手动 `kubectl apply` 修改。任何绕过 Git 直接在集群上做的变更（包括 `kubectl apply`、`kubectl edit` 等）都会造成 Git 与集群状态不一致。如需变更，应先更新 Git，再让 Argo CD 同步

[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)

[stackoverflow.com](https://stackoverflow.com)

。

移除元数据字段后恢复同步。正如经验所示，手动编辑并删除元数据中这类字段后，Argo CD 会检测到状态一致，从而显示为 `InSync`。这验证了这些字段本身并不是声明性配置的一部分，只要不在 Git 清单中出现即可。特别提醒：始终在 Git 中只保留 `metadata.name`、`namespace`、`labels` 等必要项，其他字段由集群自动维护

[blog.csdn.net](https://blog.csdn.net)

[stackoverflow.com](https://stackoverflow.com)

。

参考资料：Argo CD 官方文档  
[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)  
、Kubernetes 配置示例  
[blog.csdn.net](https://blog.csdn.net)  
、相关社区讨论  
[stackoverflow.com](https://stackoverflow.com)  
[stackoverflow.com](https://stackoverflow.com)  
等。



好的，以下是轉成繁體中文的版本：

## Argo CD 同步原理

Argo CD 以 Git 儲存庫中的宣告式資源清單作為「期望狀態」，定期對照 Kubernetes 叢集中的「實際狀態」進行比對。若兩者有任何差異，Argo CD UI 就會顯示 **OutOfSync**（不同步），否則顯示 **InSync**。官方文件指出，如果清單（manifest）中包含了實際狀態中不存在或多餘的欄位，這些欄位在查詢叢集資源時會被丟棄，從而偵測到差異並標示 OutOfSync【[argo-cd.readthedocs.io](https://argo-cd.readthedocs.io)】。換言之，Argo CD 偵測差異的邏輯是針對【Git 清單 ↔ 叢集實際物件】進行的三路合併比較，一旦發現 Git 清單與叢集物件不同，就會標記為 OutOfSync。

## 為什麼 `kubectl apply` 會導致 OutOfSync





- **手動 `kubectl apply` 與 GitOps 衝突**

在 Argo CD 管理下的資源，如果你直接在叢集上用 `kubectl apply` 手動更新，而沒有同步回 Git，就會改變實際物件狀態卻不更新 Git 清單，Argo CD 會將此視為「漂移」。Argo CD 預設使用 `kubectl apply` 的三方合併模式，同步時會在叢集物件的

`kubernetes.io/last-applied-configuration` 註解中記錄上次套用的狀態【stackoverflow.com】。若手動套用的 YAML 與 Git 中的宣告不一致，Argo 就會發現不同並回報 OutOfSync。

- **中繼資料 (metadata) 欄位衝突**

從叢集匯出的 YAML 通常包含大量中繼資料欄位（如 `creationTimestamp`、`resourceVersion`、`uid`、`generation` 等）。這些欄位都是 Kubernetes 自動產生或內部使用的，並不屬於使用者宣告的內容。若將含有這些欄位的 YAML 再次用 `kubectl apply`，就會觸發合併衝突。例如有討論指出，`kubectl apply` 會基於本地檔案、叢集即時狀態以及 `last-applied-configuration` 註解執行三向合併【stackoverflow.com】。若 `last-applied-configuration` 中意外包含 `resourceVersion` 等欄位，而本地檔案沒有，對比時就會嘗試將該欄位從叢集物件刪除，導致衝突錯誤【stackoverflow.com】。這種衝突也會讓 Argo CD 偵測到「叢集物件與 Git 清單不一致」，因此顯示 OutOfSync。官方工具的錯誤訊息類似：

```
Operation cannot be fulfilled on <kind>/<name>: the object has been
modified; please apply your changes to the latest version
```

這表示 YAML 中的叢集中繼資料（如 `resourceVersion`）與當前物件不符，必須刪除後再重試【stackoverflow.com】。

- **範例流程**

當執行 `kubectl apply -f file.yaml` 時，如果 `file.yaml` 是透過 `kubectl get -o yaml` 匯出的原始物件，內含上述中繼資料欄位，那麼 `apply` 會嘗試移除這些欄位，進而引發錯誤或部分忽略操作。即使沒有明顯錯誤，最終叢集中的資源狀態（特別是中繼資料）也會與 Git 儲存的清單不同，導致 Argo CD 報告 OutOfSync。



# 為什麼 `kubectl edit` 刪除中繼資料欄位後會恢復同步

當你在叢集上用 `kubectl edit` 手動刪除那些自動產生的中繼資料欄位後，叢集物件的實際狀態就與 Git 清單的描述完全一致了。因為 Git 清單通常**不包含**這些欄位（它們是叢集自行管理的結果），所以刪除後 Argo CD 再次比較時不會偵測到差異，狀態就會恢復為 **InSync**。

- **對比校驗**

舉例來說，如果 Git 清單只描述了 Pod 的 `spec`，而不包含

`metadata.creationTimestamp`、`metadata.resourceVersion`、`metadata.uid`、`metadata.generation` 等欄位，那麼叢集物件實際上也不「期望」這些欄位有固定值。移除這些欄位後，叢集中物件的剩餘內容（名稱、標籤、spec 等）就與 Git 清單一致，Argo CD 就不會再發現多餘或缺少欄位，因而恢復同步。

- **Argo CD 差異檢測邏輯**

如前所述，Argo CD 對比的是 Git 清單與「查詢到的叢集物件」。若叢集物件在 `metadata` 部分僅保留名稱、標籤、命名空間等必要欄位，而不帶有動態變動的欄位（如 `resourceVersion`），Argo 的比對就不會認為有差異。換句話說，**手動移除中繼資料中的動態欄位，讓宣告式描述（Git）與實際狀態保持一致，就能解決 OutOfSync。**

---

## 是否可以移除 `metadata` 下的欄位

- **欄位性質**

`metadata.creationTimestamp`、`metadata.resourceVersion`、`metadata.uid`、`metadata.generation` 等欄位都是由 Kubernetes 控制平面自動維護的內部資訊。它們不是使用者應該在 YAML 清單中指定的內容，而是叢集建立或更新物件時自動產生的。依照最佳實務，這些欄位應該在宣告式配置中**省略**。Kubernetes 官方指南也指出，中繼資料下有些鍵並非必要，例如 `creationTimestamp` 與 `selfLink` 可刪除【[blog.csdn.net](https://blog.csdn.net)】。使用者只需要在 YAML 中保留必要的 `name`、`namespace`、`labels` 等，其餘欄位應交由叢集管理。

- **安全移除**

從 YAML 清單中刪除這些欄位是安全且推薦的做法。正如文件與社群經驗所示，這些欄位一旦被刪除，叢集在建立或更新時會自動補上；反之若清單硬是包含它們，就可能引發衝突

【stackoverflow.com】。有討論者總結：「刪除 ``spec.clusterIP``、``metadata.uid`` 等叢集特有欄位後再部署，並不會有影響；這些欄位都是叢集自動生成或維護的」

【stackoverflow.com】。

**結論：可以而且應該移除這些欄位**，讓 Git 清單僅描述真正的宣告內容，由 Kubernetes 控制器自行管理內部中繼資料。

- **Argo CD 對比行為**

在 Argo CD 的比對機制中，``resourceVersion`` 和 ``generation`` 等欄位通常不被視為宣告式的一部分（Git 中不存在）。事實上，若 Git 清單沒有它們，Argo CD 在計算 diff 時不會要求將這些欄位「補回」。相反地，如果清單裡出現了這些欄位，Argo CD 就會把它們識別為多餘欄位並報告差異【argo-cd.readthedocs.io】。因此，最簡單的作法是：**讓 Git 清單完全不包含這些欄位**，這樣 Argo CD 就不會將它們當作同步目標。

---

## 小結

- **不要將叢集自動填充的中繼資料帶入 Git 清單。**例如透過 ``kubectl get -o yaml`` 匯出的完整物件通常包含 ``creationTimestamp``、``resourceVersion``、``uid``、``generation`` 等欄位，這些只在當前叢集狀態有效。若複製到 Git 儲存庫，就會讓 Argo CD 視為多餘欄位並觸發 OutOfSync【argo-cd.readthedocs.io】【blog.csdn.net】。
- **使用 GitOps 時避免手動 ``kubectl apply`` 修改。**任何繞過 Git 直接在叢集做的變更（包括 ``kubectl apply``、``kubectl edit`` 等）都會造成 Git 與叢集狀態不一致。如需修改，應先更新 Git，再由 Argo CD 同步【argo-cd.readthedocs.io】【stackoverflow.com】。
- **刪除中繼資料欄位後能恢復同步。**如同經驗所示，手動編輯並刪除這些欄位後，Argo CD 偵測到狀態一致，Sync 就會顯示為 **InSync**。這也驗證了這些欄位本質上並不是宣告式配置的一部分，只要不出現在 Git 清單中即可。特別提醒：**Git 中只保留 ``metadata.name``、``namespace``、``labels`` 等必要欄位，其餘欄位交由叢集自動維護**【blog.csdn.net】【stackoverflow.com】。

---

📖 參考資料：

- Argo CD 官方文件 [【argo-cd.readthedocs.io】](https://argo-cd.readthedocs.io)
- Kubernetes 配置示例 [【blog.csdn.net】](https://blog.csdn.net)
- 社群討論與經驗 [【stackoverflow.com】](https://stackoverflow.com)

---

要不要我幫你整理一個簡短的 **對照表**（哪些欄位建議保留、哪些一定要刪掉），方便直接套用在 GitOps 的 YAML？



對照表（哪些欄位建議保留、哪些一定要刪掉）



很好 👍 你問到的其實就是「**kubectl get -o yaml** 匯出的物件」→「能被 **GitOps** 工具（**Argo CD**、**Flux**）安全追蹤的 **YAML**」轉換規則。

以下整理一份 **對照表**，幫你判斷哪些欄位要保留、哪些要刪掉，避免 sync 差異一直存在：

---

## ✅ 建議保留欄位

區塊/欄位	說明
<code>`apiVersion`</code>	必須保留，物件版本
<code>`kind`</code>	必須保留，物件種類
<code>`metadata.name`</code>	必須保留，物件名稱
<code>`metadata.namespace`</code>	若非 default，建議保留
<code>`metadata.labels`</code>	可保留，若應用邏輯需要
<code>`metadata.annotations`</code> （部分）	只保留業務或 Helm/Argo 需要的，例如 <code>`argocd.argoproj.io/sync-wave`</code> 、Helm release 的 annotation
<code>`spec.*`</code>	全部保留，因為定義 desired state
<code>`data`</code> / <code>`stringData`</code>	ConfigMap / Secret 的內容（必要時）
<code>`rules`</code> 、 <code>`subjects`</code>	RBAC 類物件的定義
<code>`ports`</code> 、 <code>`selector`</code> 、 <code>`template`</code>	Deployment/Service/Pod 重要欄位

## ❌ 建議刪除欄位（避免 drift）

區塊/欄位	說明
<code>`metadata.uid`</code>	Kubernetes runtime 自動產生
<code>`metadata.resourceVersion`</code>	每次更新都會變，不能放進 Git
<code>`metadata.generation`</code>	系統自動遞增
<code>`metadata.creationTimestamp`</code>	物件建立時間
<code>`metadata.managedFields`</code>	K8s 1.18+ 新增，用來記錄誰改了哪些欄位

區塊/欄位	說明
<code>`metadata.annotations.kubectl.kubernetes.io/last-applied-configuration`</code>	kubectl apply --save-config 產生，通常要刪掉
<code>`status.*`</code>	runtime 狀態 (Pod 狀態、Service endpoint、Replica 狀態)，全部要移除

## ⚠ 視情況決定

區塊/欄位	說明
<code>`metadata.annotations`</code>	如果是 Argo/Helm 相關的就要留，其它像 auto scaling、Prometheus operator 產生的要評估
<code>`metadata.labels`</code>	CI/CD pipeline 或監控可能會動態加 label，要決定要不要納管
<code>`spec.replicas`</code>	有些團隊會用 HPA 自動調整副本數 → 這種情況下 replicas 可以不要放進 Git，由 HPA 控制

📌 簡單來說：

- 保留 → **desired state (spec, config, labels/annotations if business needed)**
- 刪掉 → **runtime state (status, uid, resourceVersion, timestamps, managedFields)**

要不要我幫你做一個 **範例對照檔 (Deployment.yaml 前後對比)**，讓你一看就知道哪些要砍、哪些要留？



範例對照檔 (Deployment.yaml 前後對比)



已思考 6s

- 我們需要展示部署檔案（`Deployment.yaml`）的原始與已清理版本。使用 Markdown 並以 `#` 標註區段標題。首先，提供包含元資料欄位的原始 `kubectl get -o yaml` 範例，然後提供一個能清理的版本。

\*\*原始

☑ 完成

## Deployment 範例 — 前後對比（繁體中文）

下面用一個你貼過的命名（`ifond` / `f18-vmvminsert-virtual-svr-80`）做示例。左邊是從 `kubectl get -o yaml` 直接匯出的 **raw**（含大量 **cluster-managed metadata**），右邊是可以直接放進 **Git** 並由 **Argo CD** 管理的「乾淨版」（已刪除所有 runtime / server-managed 欄位，只保留必要的宣告內容）。

### 原始（kubectl get -o yaml） — 不要把這個放進 Git

```
apiVersion: apps/v1
kind: Deployment
metadata:
```



```
name: f18-vmvminsert-virtual-svr-80
namespace: ifond
labels:
  app.kubernetes.io/instance: ifond-vmvminsert-prod-fab18edge-fz
  app.kubernetes.io/name: vminsert
uid: 94445b8-8926-4baf-a6f7-5a4767b06ce7
resourceVersion: "190657607"
generation: 7
creationTimestamp: "2025-03-21T06:20:07Z"
managedFields:
- manager: kubectl
  operation: Update
  apiVersion: apps/v1
  time: "2025-03-21T06:20:07Z"
annotations:
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"name":"f18-
vmvminsert-virtual-svr-80","namespace":"ifond"},"spec":{"replicas":3}}
spec:
  replicas: 3
  selector:
    matchLabels:
      app: vminsert
  template:
    metadata:
      labels:
        app: vminsert
    spec:
      containers:
      - name: vminsert
        image: myrepo/vminsert:1.2.3
        ports:
        - containerPort: 80
status:
  observedGeneration: 7
  replicas: 3
  updatedReplicas: 3
  readyReplicas: 3
  availableReplicas: 3
  conditions:
  - type: Available
```

```
status: "True"
lastUpdateTime: "2025-03-21T06:25:00Z"
lastTransitionTime: "2025-03-21T06:22:00Z"
reason: MinimumReplicasAvailable
message: Deployment has minimum availability.
```

“問題點：上面 `uid`、`resourceVersion`、`generation`、`creationTimestamp`、`managedFields`、`annotations.kubect1.kubernetes.io/last-applied-configuration`、`status` 都是由 Kubernetes 叢集/控制平面維護或 runtime 產生的值。把它們放入 Git 會導致 Argo CD / kubectl apply 的 diff 與合併行為出現不必要的差異或衝突。”

## 乾淨版（可 Commit 到 Git / 由 Argo CD 管理）

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: f18-vmvminsert-virtual-svr-80
  namespace: ifond
  labels:
    app.kubernetes.io/instance: ifond-vmvminsert-prod-fab18edge-fz
    app.kubernetes.io/name: vminsert
spec:
  replicas: 3
  selector:
    matchLabels:
      app: vminsert
  template:
    metadata:
      labels:
        app: vminsert
    spec:
      containers:
        - name: vminsert
          image: myrepo/vminsert:1.2.3
```



```
ports:
- containerPort: 80
```

“變更說明：刪除了 `.metadata.uid``、`.metadata.resourceVersion``、`.metadata.generation``、`.metadata.creationTimestamp``、`.metadata.managedFields``、`.metadata.annotations.kubectl.kubernetes.io/last-applied-configuration`` 以及整個 `.status`` 區塊。保留 `name``、`namespace``、`labels``、`spec`` (即宣告式 *desired state*)。”

## 哪些欄位被移除 & 為什麼（快速對照）

- `.metadata.uid`` → 由 API server 產生，代表物件唯一 ID（不應納入宣告）
- `.metadata.resourceVersion`` → 每次更新都變，供 k8s 用於快取/一致性檢查（不應納入宣告）
- `.metadata.generation`` → controller 用來追蹤 spec 版本，會自動遞增
- `.metadata.creationTimestamp`` → 物件建立時間（純資訊）
- `.metadata.managedFields`` → 1.18+ 的欄位，紀錄哪些 field 被誰管理（會造成大量 non-deterministic diff）
- `.metadata.annotations.kubectl.kubernetes.io/last-applied-configuration`` → kubectl apply 的內部註記（會干擾三向合併）
- `.status.*`` → runtime 狀態（Pod 狀態、可用副本數、Conditions 等），不屬於 *desired state*



# 自動化清理指令 (CI / pre-commit / pipeline 可用)

## 使用 `yq` (Mike Farah 的 yq v4)

```
kubectl get deployment f18-vmvminsert-virtual-svr-80 -n ifond -o yaml \
  | yq eval 'del(.metadata.creationTimestamp, .metadata.resourceVersion,
.metadata.uid, .metadata.generation, .metadata.managedFields,
.metadata.annotations."kubectl.kubernetes.io/last-applied-configuration",
.status)' - \
  > deployment.cleaned.yaml
```

## 使用 `kubectl` + `jq` (JSON) 再轉回 YAML (若系統沒 yq)

```
kubectl get deployment f18-vmvminsert-virtual-svr-80 -n ifond -o json \
  | jq 'del(.metadata.creationTimestamp, .metadata.resourceVersion,
.metadata.uid, .metadata.generation, .metadata.managedFields,
.metadata.annotations."kubectl.kubernetes.io/last-applied-configuration",
.status)' \
  > deployment.cleaned.json
```

# 轉為 yaml (若安裝 yq 可用)

```
cat deployment.cleaned.json | yq eval -P - > deployment.cleaned.yaml
```

## 一個簡單的 CI step (GitLab CI 範例)

```
stages:
  - prepare

clean-manifest:
  stage: prepare
  script:
    - kubectl get deployment f18-vmvminsert-virtual-svr-80 -n ifond -o yaml >
raw.yaml
    - yq eval 'del(.metadata.creationTimestamp, .metadata.resourceVersion,
```

```
.metadata.uid, .metadata.generation, .metadata.managedFields,  
.metadata.annotations."kubect1.kubernetes.io/last-applied-configuration",  
.status)' raw.yaml > manifest-for-git.yaml  
- git add manifest-for-git.yaml && git commit -m "chore: update cleaned  
manifest" || echo "no changes"
```

## 小技巧與注意事項

- 別把 ``kubectl get -o yaml`` 的 **raw** 直接 **commit**。先用上述清理流程去掉 runtime 欄位。
- 保留必要 **annotations/labels**（例如 Helm 的 release annotation、ArgoCD 需要的 annotation）——這些可以放在白名單中，不要一律刪掉。
- 如果你有 **controller**（例如 HPA）會改 ``spec.replicas``，就評估是否要把 ``replicas`` 從 **Git** 中移除，或讓 **Git** 為單一真相並接受 **controller** 的行為（視團隊策略而定）。
- 在 **CI** 裡自動清理是常見作法：從 cluster 抓回來做檢查或建立 PR 時，先清理再比較/commit。

要不要我幫你把剛剛你原始貼的那段不整齊的 YAML（你之前貼的有換行亂碼）清理一次，並回傳一個可以直接 commit 的 ``deployment.cleaned.yaml``？我可以直接把 ``uid`` / ``resourceVersion`` / ``creationTimestamp`` / ``generation`` / ``managedFields`` / ``status`` 都刪掉，然後把結果貼給你。