Die vier fundamentalen Objekttypen in Git □□

Git ist im Kern nichts anderes als eine **inhaltsadressierte Datenbank** – ein elegantes System, das auf nur vier fundamentalen Objekttypen basiert. Wenn du verstehst, wie diese Objekte funktionieren und zusammenhängen, wirst du Git nie wieder als „magische Black Box" wahrnehmen, sondern als das durchdachte, transparente System, das es wirklich ist.

Das Fundament: Gits Objektmodell

Jedes Git-Repository speichert seine gesamte Geschichte in Form von **Objekten** im Verzeichnis `.git/objects/`. Jedes Objekt wird durch einen **SHA-1-Hash** identifiziert – eine 40-stellige hexadezimale Zeichenkette, die aus dem Inhalt des Objekts berechnet wird. Das bedeutet: *Gleicher Inhalt ergibt immer den gleichen Hash*, egal auf welchem Computer oder in welchem Repository.

```
flowchart TB
    subgraph Objektdatenbank[".git/objects/"]
        B1["Blob\nfa3c2..."]
        B2["Blob\n8b137..."]
        B3["Blob\ne69de..."]
        T1["Tree\na1b2c..."]
        T2["Tree\nd4e5f..."]
        C1["Commit\n7f8d9..."]
        C2["Commit\n3a4b5..."]
        TAG["Tag\nc6d7e..."]
    end

    C1 --> T1
    C2 --> T2
    C2 -.->|parent| C1
    T1 --> B1
    T1 --> B2
    T1 --> T2
```

```
    T2 --> B3
    TAG --> C2

    style B1 fill:#e8f4e8,color:#000000
    style B2 fill:#e8f4e8,color:#000000
    style B3 fill:#e8f4e8,color:#000000
    style T1 fill:#fff3e0,color:#000000
    style T2 fill:#fff3e0,color:#000000
    style C1 fill:#e3f2fd,color:#000000
    style C2 fill:#e3f2fd,color:#000000
    style TAG fill:#fce4ec,color:#000000
```

---

1. Blob – Der Dateiinhalt □□

Ein **Blob** (Binary Large Object) ist das einfachste Git-Objekt. Es speichert ausschließlich den **Inhalt einer Datei** – nichts weiter. Kein Dateiname, keine Berechtigungen, keine Metadaten. Nur der pure Inhalt.

Eigenschaften des Blob-Objekts

- Speichert **rohen Dateiinhalt** (Text oder Binärdaten)
- Enthält **keinen Dateinamen** – der Name wird im Tree gespeichert
- Enthält **keine Zeitstempel** oder andere Metadaten
- **Identische Dateien** erzeugen identische Blobs (und damit denselben Hash)

Das Letztere ist besonders clever: Wenn du dieselbe Datei in zehn verschiedenen Verzeichnissen hast, speichert Git sie nur *einmal* als Blob. Alle Trees verweisen dann auf denselben Blob.

Blob mit `git cat-file` untersuchen

Lass uns das praktisch ausprobieren. Erstelle zunächst ein Test-Repository:

```
# Neues Repository erstellen
mkdir git-objekte-demo && cd git-objekte-demo
git init

# Eine einfache Datei erstellen
echo "Hallo Git-Welt!" > hallo.txt
```

```
git add hallo.txt
```

Mit `git add` wurde die Datei in die Staging Area aufgenommen – und dabei wurde ein Blob-Objekt erstellt. Finde den Hash heraus:

```
# Hash des Blobs anzeigen (aus dem Index/Staging Area)
git ls-files -s
```

Ausgabe:

```
100644 d6a96ae3b442218a91512b9e1c57b9578b8998e8 0□hallo.txt
```

Die Zahl `100644` sind die Dateiberechtigungen, `d6a96ae...` ist der SHA-1-Hash des Blobs. Jetzt können wir den Blob untersuchen:

```
# Typ des Objekts anzeigen
git cat-file -t d6a96ae3b442218a91512b9e1c57b9578b8998e8
```

Ausgabe:

```
blob
```

```
# Inhalt des Blobs anzeigen
git cat-file -p d6a96ae3b442218a91512b9e1c57b9578b8998e8
```

Ausgabe:

```
Hallo Git-Welt!
```

```
# Größe des Blobs in Bytes
git cat-file -s d6a96ae3b442218a91512b9e1c57b9578b8998e8
```

Ausgabe:

```
16
```

> □□ **Tipp:** Du musst nicht den vollständigen Hash eingeben. Git akzeptiert auch eindeutige Präfixe, z.B. `git cat-file -p d6a96ae` oder sogar `git cat-file -p d6a96`.

---

2. Tree – Das Verzeichnis □□

Ein **Tree** repräsentiert ein **Verzeichnis** in Git. Er ist quasi das „Inhaltsverzeichnis", das Dateinamen mit Blobs verknüpft. Ein Tree enthält eine Liste von Einträgen, wobei jeder Eintrag aus folgenden Komponenten besteht:

- **Modus** (Dateiberechtigungen, z.B. `100644` für normale Dateien, `100755` für ausführbare Dateien, `040000` für Unterverzeichnisse)
- **Objekttyp** (blob oder tree)
- **SHA-1-Hash** des referenzierten Objekts
- **Name** der Datei oder des Unterverzeichnisses

Tree-Struktur visualisiert

```
flowchart TB
    ROOT["Tree: Wurzelverzeichnis\na1b2c3d..."]

    ROOT --> E1["100644 blob fa3c2...\nhallo.txt"]
    ROOT --> E2["100644 blob 8b137...\nREADME.md"]
    ROOT --> E3["040000 tree d4e5f...\nsrc/"]

    SRC["Tree: src/\nd4e5f6g..."]
    E3 -.-> SRC

    SRC --> E4["100644 blob e69de...\nindex.php"]
    SRC --> E5["100755 blob 12345...\nscript.sh"]

    style ROOT fill:#fff3e0,color:#000000
    style SRC fill:#fff3e0,color:#000000
    style E1 fill:#e8f4e8,color:#000000
    style E2 fill:#e8f4e8,color:#000000
    style E3 fill:#fff3e0,color:#000000
    style E4 fill:#e8f4e8,color:#000000
    style E5 fill:#e8f4e8,color:#000000
```

Tree praktisch untersuchen

Erweitern wir unser Demo-Repository und erstellen einen Commit, damit wir Trees untersuchen können:

```
# Weitere Dateien und ein Verzeichnis erstellen
echo "# Mein Projekt" > README.md
mkdir src
echo "<?php echo 'Hello World';" > src/index.php
git add .
git commit -m "Erster Commit mit Projektstruktur"
```

Jetzt können wir den Tree des letzten Commits anschauen:

```
# Den Tree-Hash des letzten Commits finden
git cat-file -p HEAD
```

Ausgabe:

```
tree 4b825dc642cb6eb9a060e54bf8d69288fbee4904
author Max Mustermann <max@example.com> 1703847600 +0100
committer Max Mustermann <max@example.com> 1703847600 +0100

Erster Commit mit Projektstruktur
```

Den Tree-Hash (hier beginnt er mit `4b825dc...`, dein Hash wird anders sein) können wir nun untersuchen:

```
# Tree-Inhalt anzeigen (ersetze den Hash durch deinen)
git cat-file -p 4b825dc642cb6eb9a060e54bf8d69288fbee4904
```

Ausgabe:

```
100644 blob d6a96ae3b442218a91512b9e1c57b9578b8998e8□hallo.txt
100644 blob 7e59600739c96546163833214c36459e324bad0a□README.md
040000 tree 8f94139338f9404f26296befa88755fc2598c289□src
```

Du siehst: Der Tree enthält Verweise auf zwei Blobs (die Dateien) und einen weiteren Tree (das `src`-Verzeichnis). Lass uns auch den Unter-Tree anschauen:

```
# Unter-Tree (src/) untersuchen
git cat-file -p 8f94139338f9404f26296befa88755fc2598c289
```

Ausgabe:

```
100644 blob 8b137891791fe96927ad78e64b0aad7bded08bdc□index.php
```

Praktischer Shortcut mit `git ls-tree`

Für die Untersuchung von Trees gibt es auch den spezialisierten Befehl `git ls-tree`:

```
# Tree des aktuellen Commits rekursiv anzeigen
git ls-tree -r HEAD
```

Ausgabe:

```
100644 blob d6a96ae3b442218a91512b9e1c57b9578b8998e8□hallo.txt
100644 blob 7e59600739c96546163833214c36459e324bad0a□README.md
100644 blob 8b137891791fe96927ad78e64b0aad7bded08bdc□src/index.php
```

---

3. Commit – Der Schnappschuss □□

Ein **Commit** ist das Herzstück von Git. Er repräsentiert einen **Schnappschuss** deines gesamten Projekts zu einem bestimmten Zeitpunkt. Wichtig zu verstehen: Ein Commit speichert *nicht* die Änderungen (Diffs), sondern verweist auf einen Tree, der den **kompletten Zustand** aller Dateien beschreibt.

Bestandteile eines Commit-Objekts

Feld	Beschreibung
`tree`	SHA-1-Hash des Root-Trees (Projekt-Schnappschuss)
`parent`	SHA-1-Hash des Vorgänger-Commits (kann fehlen beim ersten Commit, oder mehrfach vorhanden sein bei Merge-Commits)
`author`	Name, E-Mail und Zeitstempel der Person, die die Änderung ursprünglich erstellt hat
`committer`	Name, E-Mail und Zeitstempel der Person, die den Commit erstellt hat (kann bei Cherry-Picks/Rebases abweichen)
Leerzeile	Trennt Header von der Nachricht
Commit-Message	Die Beschreibung der Änderung

Commit praktisch untersuchen

```
# Einen zweiten Commit erstellen
echo "Neue Zeile" >> hallo.txt
git add hallo.txt
git commit -m "hallo.txt erweitert"

# Die letzten beiden Commits anzeigen
git log --oneline -2
```

Ausgabe:

```
a7b3c9d (HEAD -> main) hallo.txt erweitert
f1e2d3c Erster Commit mit Projektstruktur
```

Jetzt untersuchen wir den neuesten Commit:

```
git cat-file -p HEAD
```

Ausgabe:

```
tree 9f8e7d6c5b4a3210fedcba9876543210abcdef12
parent f1e2d3c4a5b6c7d8e9f0a1b2c3d4e5f6a7b8c9d0
author Max Mustermann <max@example.com> 1703851200 +0100
committer Max Mustermann <max@example.com> 1703851200 +0100

hallo.txt erweitert
```

Du siehst hier den **parent**-Eintrag, der auf den vorherigen Commit verweist. Diese Verkettung durch Parent-Referenzen bildet die **Commit-Historie** – eine einfach verkettete Liste (oder bei Merges: ein gerichteter azyklischer Graph).

Die Commit-Kette visualisiert

```
flowchart RL
    C3["Commit C3\na7b3c9d\n\nhallo.txt erweitert"]
    C2["Commit C2\nf1e2d3c\n\nErster Commit"]
    C1["Commit C1\n0a1b2c3\n\nInitial commit"]
```

```
    C3 -->|parent| C2
    C2 -->|parent| C1

    HEAD["HEAD"]
    MAIN["main"]

    HEAD -.-> MAIN
    MAIN -.-> C3

    style C3 fill:#e3f2fd,color:#000000
    style C2 fill:#e3f2fd,color:#000000
    style C1 fill:#e3f2fd,color:#000000
    style HEAD fill:#ffecb3,color:#000000
    style MAIN fill:#c8e6c9,color:#000000
```

Merge-Commits haben mehrere Parents

Bei einem Merge-Commit gibt es mehrere `parent`-Einträge:

```
# Beispiel eines Merge-Commits (nach einem Merge erstellt)
git cat-file -p <merge-commit-hash>
```

Ausgabe:

```
tree 1234567890abcdef...
parent aaaaaaa111111...   <-- erster Parent (der Branch, IN den gemerged wurde)
parent bbbbbbb222222...   <-- zweiter Parent (der Branch, DER gemerged wurde)
author Max Mustermann <max@example.com> 1703854800 +0100
committer Max Mustermann <max@example.com> 1703854800 +0100

Merge branch 'feature/login'
```

---

4. Tag – Die benannte Markierung □□

Ein **Tag** ist ein benannter Zeiger auf einen bestimmten Commit – typischerweise verwendet für Release-Versionen wie `v1.0.0` oder `v2.3.1`. Git kennt zwei Arten von Tags:

Lightweight Tags vs. Annotated Tags

Eigenschaft	Lightweight Tag	Annotated Tag
Speicherung	Nur eine Referenz in `.git/refs/tags/`	Eigenes Tag-Objekt in `.git/objects/`
Metadaten	Keine	Tagger, Datum, Nachricht
GPG-Signatur	Nicht möglich	Möglich
Empfehlung	Für temporäre/lokale Markierungen	Für Releases und wichtige Meilensteine

Annotated Tag erstellen und untersuchen

```
# Annotated Tag erstellen
git tag -a v1.0.0 -m "Erste stabile Version"

# Tag-Objekt untersuchen
git cat-file -t v1.0.0
```

Ausgabe:

```
tag
```

```
git cat-file -p v1.0.0
```

Ausgabe:

```
object a7b3c9d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
type commit
tag v1.0.0
tagger Max Mustermann <max@example.com> 1703858400 +0100

Erste stabile Version
```

Das Tag-Objekt enthält:

- **object**: Der SHA-1-Hash des Commits, auf den der Tag zeigt
- **type**: Der Typ des referenzierten Objekts (normalerweise `commit`)

- **tag**: Der Name des Tags
- **tagger**: Wer den Tag wann erstellt hat
- **Nachricht**: Die Tag-Beschreibung

Lightweight Tag zum Vergleich

```
# Lightweight Tag erstellen
git tag v1.0.0-beta

# Typ prüfen - zeigt direkt auf den Commit!
git cat-file -t v1.0.0-beta
```

Ausgabe:

```
commit
```

Ein Lightweight Tag ist kein eigenes Objekt, sondern nur eine Referenz-Datei. Wenn du `git cat-file -p v1.0.0-beta` ausführst, siehst du direkt den Commit-Inhalt, nicht ein Tag-Objekt.

Wie alles zusammenhängt □□

Jetzt, wo du alle vier Objekttypen kennst, lass uns das große Ganze betrachten. Ein typisches Git-Repository mit zwei Commits könnte intern so aussehen:

```
flowchart TB
    subgraph Referenzen
        HEAD["HEAD\n→ refs/heads/main"]
        MAIN["refs/heads/main\n→ commit 2"]
        TAG["refs/tags/v1.0.0\n→ tag-objekt"]
    end

    subgraph Objekte
        TAGO["Tag-Objekt\nv1.0.0\n→ commit 2"]

        C2["Commit 2\ntree: T2\nparent: C1"]
        C1["Commit 1\ntree: T1\nparent: keine"]

        T2["Tree T2\nhallo.txt → B2\nREADME.md → B3\nsrc/ → T3"]
```

```
    T1["Tree T1\nhallo.txt → B1\nREADME.md → B3"]
    T3["Tree T3\nindex.php → B4"]

    B1["Blob B1\nHallo Git-Welt!"]
    B2["Blob B2\nHallo Git-Welt!\nNeue Zeile"]
    B3["Blob B3\n# Mein Projekt"]
    B4["Blob B4\nPHP-Code..."]
end

HEAD -.-> MAIN
MAIN -.-> C2
TAG -.-> TAGO
TAGO --> C2

C2 --> T2
C2 -->|parent| C1
C1 --> T1

T2 --> B2
T2 --> B3
T2 --> T3
T1 --> B1
T1 --> B3
T3 --> B4

style HEAD fill:#ffecb3,color:#000000
style MAIN fill:#c8e6c9,color:#000000
style TAG fill:#c8e6c9,color:#000000
style TAGO fill:#fce4ec,color:#000000
style C1 fill:#e3f2fd,color:#000000
style C2 fill:#e3f2fd,color:#000000
style T1 fill:#fff3e0,color:#000000
style T2 fill:#fff3e0,color:#000000
style T3 fill:#fff3e0,color:#000000
style B1 fill:#e8f4e8,color:#000000
style B2 fill:#e8f4e8,color:#000000
style B3 fill:#e8f4e8,color:#000000
style B4 fill:#e8f4e8,color:#000000
```

Beachte die Effizienz: Die Datei `README.md` hat sich zwischen Commit 1 und Commit 2 nicht geändert. Daher verweisen beide Trees auf **denselben Blob** (B3). Git speichert den Inhalt nur einmal!

Die wichtigsten `git cat-file`-Optionen im Überblick

Option	Beschreibung	Beispiel
`-t`	Zeigt den **Typ** des Objekts	`git cat-file -t HEAD` → `commit`
`-p`	**Pretty-Print**: Zeigt den Inhalt formatiert an	`git cat-file -p HEAD`
`-s`	Zeigt die **Größe** in Bytes	`git cat-file -s HEAD`
`-e`	**Existenz-Check**: Exit-Code 0 wenn Objekt existiert	`git cat-file -e abc123`

Nützliche Kombinationen und Shortcuts

```
# Alle Objekte im Repository auflisten
find .git/objects -type f | head -20

# Typ und Größe aller Objekte eines Commits anzeigen
git rev-list --objects HEAD | while read hash name; do
  echo "$hash $(git cat-file -t $hash) $(git cat-file -s $hash) $name"
done

# Direkter Zugriff auf Tree eines Commits
git cat-file -p HEAD^{tree}

# Direkter Zugriff auf eine Datei in einem Commit
git cat-file -p HEAD:src/index.php
```

Praktische Übung zum Selbermachen □□

Um das Gelernte zu festigen, probiere Folgendes in einem Test-Repository:

1. **Erstelle ein neues Repository** mit einigen Dateien und Unterverzeichnissen
2. **Finde die Blob-Hashes** deiner Dateien mit `git ls-files -s`
3. **Untersuche den Tree** deines letzten Commits mit `git cat-file -p HEAD^{tree}`
4. **Vergleiche zwei Commits**: Erstelle eine kleine Änderung, committe sie, und vergleiche die Tree-Hashes beider Commits. Welche Blobs haben sich geändert, welche sind gleich geblieben?
5. **Erstelle einen Annotated Tag** und untersuche sein Objekt

Mit diesem Wissen über Git-Interna bist du bestens vorbereitet für die fortgeschrittenen Themen wie Rebase, Reflog und Troubleshooting – denn du verstehst jetzt, *was* Git eigentlich tut, wenn du diese Befehle ausführst! □□

Revision #2
Created 2026-06-29 15:10:19 UTC by art10m
Updated 2026-06-29 15:15:34 UTC by art10m