

IT 資格 道 場

GH-900 GitHub Foundations 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

GitHub Foundations (GH-900) 出題範囲 (Skills at a glance as of January 2026)

準拠

2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

この教材群の使い方(3点セット)

この試験が問うもの

第1章 GitHub と Git の基本 [1.1/1.2/1.3/1.4/1.5/1.6/1.7]

バージョン管理の目的と利点 [1.1]

集中型と分散型の違い [1.1]

Git と GitHub の違い [1.2]

ローカルとリモートのリポジトリ [1.3]

コミットとスナップショット・コミットハッシュ [1.3]

ブランチとマージ [1.3]

clone / fetch / pull / push の違い [1.3]

ステージング(add → commit) [1.3]

HEAD と既定ブランチ [1.3]

個人アカウントと Organization の違い [1.4]

Enterprise アカウントで束ねられるもの [1.4]

プラン(Free / Team / Enterprise) ごとの機能差 [1.4]

GitHub.com (Cloud) と self-hosted の選択 [1.4]

GitHub Flow [1.5]

Markdown [1.6]

GitHub Desktop・Mobile・CLI・Web の使い分け [1.7]

サンプルはここまでです

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

この試験が問うもの

GH-900 は、GitHub の基本機能と使い分けを問う試験です。開発者だけでなく、プロジェクトを管理する人や GitHub を使い始めた人も対象になります。コードを書く力ではなく、機能の名前と役割、似た機能の違い、場面に合う選び方を問います。

項目	内容
試験コード	GH-900
試験時間	100 分
合格ライン	700 点以上
受験方式	監督付きの会場またはオンライン受験 (Pearson VUE)
言語	英語・スペイン語・ポルトガル語 (ブラジル)・韓国語・日本語

分野ごとの出題比率は幅で示されます。

- 分野 1 Git と GitHub の基本: 25～30%
- 分野 2 リポジトリの操作、分野 3 GitHub での協働、分野 4 最新の開発手法、分野 6 プライバシー・セキュリティ・管理: 各 10～15%
- 分野 5 プロジェクト管理、分野 7 GitHub コミュニティ: 各 5～10%

このテキストは、この 7 分野を同じ順番で第 1 章から第 7 章まで並べています。2026 年 1 月に出題範囲が改訂され、Copilot のエージェント、Agent Mode、複数モデルへの対応、パスキー、Enterprise Managed Users が加わ

りました。試験には、よく使われるプレビュー機能の問題が出ることもあります。

機能の名前、プランの構成、無料枠や上限の数値は、GitHub が更新するたびに変わります。本文の数値は書いた時点の GitHub Docs の記述で、購入や契約の判断には使えません。試験では「どのプランのどの機能か」という関係が問われるので、数値より関係を覚えておけば足ります。

SAMPLE

第1章 GitHub と Git の基本

[1.1/1.2/1.3/1.4/1.5/1.6/1.7]

バージョン管理の目的と利点 [1.1]

バージョン管理システムは、ファイルの変更を履歴として記録する仕組みです。日常の言葉に直すと、保存のたびに「いつ、誰が、何を、なぜ変えたか」の記録が積み重なっていく仕組みです。利点は次の3つです。

- 変更履歴が残る
- 過去の状態に戻せる(復元)
- 複数人で同じファイルを分担して編集できる(共同作業)

変更履歴が残るので、不具合が出たときに、どの変更が原因かを調べられます。過去の任意の時点のファイルの状態を取り出せるので、誤った変更を元に戻せます。共同作業では、各自の変更が履歴の上でつながるので、誰がどこを変えたかを後から振り返れます。

取り違いやすい点があります。バージョン管理は、保存先の容量を広げる仕組みでも、ファイルを暗号化する仕組みでもありません。複数人が同じ箇所を編集したときの衝突も、自動では消えません。衝突は人が内容を選んで解消します。

誤って設定ファイルを書き換え、システムが動かなくなった場合を考えます。バージョン管理を使っていれば、履歴から変更の内容と時点を特定し、問題が起きる前のコミットの状態に戻せます。書き換え前のファイルが別の場所に自動で複製されるわけではありません。変更が保存される時点でテストが走り、不具合のある変更が弾かれることもありません。テストの自動実行は GitHub Actions の役割で、権限の管理は別の仕組みです。

戻し方は次の 2 通りです。

- 過去のコミットの内容を取り出す
- 変更を打ち消す新しいコミットを作る

共有済みの履歴では、後者が安全です。履歴を書き換えると、他の人の履歴とずれるからです。

集中型と分散型の違い [1.1]

バージョン管理の方式は、集中型と分散型の 2 つです。

集中型では、履歴を中央のサーバーが 1 か所だけで持ちます。開発者の手元にあるのは、取得した時点のファイルだけで、履歴は手元にありません。コミットのたびに中央のサーバーへつなぐ必要があり、サーバーが止まると履歴の閲覧もコミットもできません。ファイルを編集前にロックする運用をする製品もあります。

Git が採用する分散型では、各開発者が履歴全体を含む複製を手元に持ちます。リポジトリを `git clone` で取得すると、全コミットが手元に複製されるので、ネットワークがなくても、履歴の閲覧、差分の確認、コミットの作成ができます。

分散型でも、共有用のリモートリポジトリ (GitHub など) は置くのが普通です。複数人が変更を持ち寄る同期先として使うためで、Git の仕組みとして必須ではありません。履歴が置かれるのはサーバーだけではなく、各開発者の手元にもあります。

分散型の効果は、障害のときに表れます。共有サーバーのディスク障害でリモートのデータが失われても、メンバーの手元のクローンは履歴全体を含んでいるので、最新に近いクローンを新しいリモートへ `git push` すれば復旧でき

ます。クローンが持つのは最後に取得した時点までの内容なので、未取得のコミットは、その持ち主の手元から集めます。変更の一覧だけが手元に残る仕組みではありません。

共有サーバーが数時間止まったときも、Git のチームは手元の履歴を使った過去のコミットの閲覧や差分の確認と、コミットの作成を続けられます。手元に取得していないブランチの履歴は、手元に無いので見られません。他のメンバーの未 push の変更を取り込むことも、自分のコミットを他の人から見える状態にすることも、サーバーが必要です。サーバーが戻ったあとで push すれば、手元で積んだコミットがそのまま共有されます。

Git と GitHub の違い [1.2]

Git は、手元で履歴を管理するバージョン管理のソフトウェアです。GitHub は、Git のリポジトリを預かり、共同作業の機能を加えたサービスです。

- Git: ローカルにインストールして使う。履歴の記録、ブランチ、マージなどの機能を持つ
- GitHub: Git のリポジトリのホスティングに、Issue、Pull Request、Discussions、Actions、Projects などの共同作業の機能を加えたサービス

Git は GitHub のクラウドサービスではなく、GitHub のアカウントがなくても使えます。Git と GitHub は同じソフトウェアの別名でもなく、GitHub は Git の後継のバージョン管理ソフトでもありません。GitHub の中身も Git を使っています。

インターネットにつながらない自分の PC だけで履歴を記録し始めるには、Git をインストールして、作業フォルダーで `git init` を実行します。`git init` は、

作業フォルダーに `.git` ディレクトリを作り、その中に履歴を保存します。GitHub のアカウントも、GitHub Pages も、GitHub Actions も要りません。

```
git init -b main work  
git -C work status -s
```

GitHub を併用する理由は、共有のリモトリポジトリをホストし、メンバーごとのアクセス権を設定できる点にあります。5 人のチームが各自の PC で Git を使うとき、権限はリポジトリやチーム単位で GitHub に設定します。Git 単体には、ユーザーごとの権限管理の仕組みがありません。GitHub を使っても、各メンバーの PC にコミットの履歴が保存されなくなるわけではありません。ブランチの作成やマージの操作が要らなくなるわけでもありません。Git が 2 人までの共同作業向けで、それ以上は GitHub が要る、という制約もあります。

Git は特定のサービスに依存しません。ソースコードを外部のクラウドサービスに置けない組織は、社内のサーバーに共有用の Git リポジトリを用意し、SSHなどで接続して push と pull で同期できます。共有先は URL で指定し、SSH、HTTPS、ローカルのパスなど複数の接続方式が使えます。GitHub Enterprise Server のような、自社で運用するホストも選べます。

GitHub 以外のサービスにあるリポジトリも、URL を指定して `git clone` で取得できます。GitHub のアカウントでサインインしている必要も、いったん GitHub に移す必要もなく、専用のツールも要りません。他のサービスのリポジトリを GitHub に取り込むインポートの機能は別にあり、clone するだけなら要りません。

Git の機能か GitHub の機能かは、GitHub のサーバー上にあるかどうかで切り分けます。

機能	提供元
<code>git commit</code> 、 <code>git branch</code> 、 <code>git log</code> 、 <code>git merge</code> 、 <code>git clone</code>	Git (Git 自体のコマンド)
Issue、Pull Request、Discussions、Actions、Projects、フォーク	GitHub

Pull Request のマージボタンは GitHub の機能で、ボタンの裏では GitHub が Git の仕組みでマージしています。ブランチを `git merge` でマージするのは Git の機能で、GitHub のアカウントがなくても手元のリポジトリだけで完結します。フォークは GitHub 上でリポジトリを自分のアカウントに複製する機能で、Git のコマンドは `clone` までです。Issue、Pull Request、Discussions は GitHub のサーバー上に保存されるため、`git clone` してもローカルの履歴には含まれません。

ローカルとリモートのリポジトリ [1.3]

リポジトリは、ファイルとその変更履歴を入れた入れ物です。手元の PC にあるものをローカルリポジトリ、GitHub などのサーバー上にあるものをリモートリポジトリと呼びます。

`git commit` は変更をローカルリポジトリに記録するだけで、リモートに送るのは `git push` です。`commit` はネットワークに接続してなくても実行できます。ローカルで 3 回コミットしても、`git push` をするまで、GitHub 上のリポジトリには反映されません。一定時間後に自動で同期されることも、承認されるまで保留されることもありません。手元のブランチがリモートより何コミット進んでいるかは、`git status` で確認します。

`git clone` でリモートリポジトリを複製すると、複製元のリモートに `origin` という名前が付き、`git push origin main` のようにこの名前でリモートを指定し

ます。名前は `git remote rename` で変えられ、フォークした場合は元のリポジトリを `upstream` という名前で追加するのが慣習です。

```
git remote -v
```

```
origin    /tmp/ghdemo/remote.git (fetch)
```

```
origin    /tmp/ghdemo/remote.git (push)
```

コミットとスナップショット・コミットハッシュ [1.3]

コミットは、ある時点のプロジェクトの内容(スナップショット)に、作者、日時、メッセージ、親コミットなどの情報を添えた履歴の単位です。ファイルごとの変更行だけを記録して、時点ごとの全体の状態を持たない仕組みではありません。内部では、変更のないファイルは前と同じ内容を指すので、容量は無駄に増えません。

コミットは、リモートへ送るための転送用パッケージでも、ブランチの先頭を指す名前付きの目印でもありません。ブランチの先頭を指す目印は、ブランチです。

コミットを識別する ID は、ハッシュ値です。コミットの内容と親コミットなどから計算され、リポジトリ内の他のコミットと区別できる範囲で先頭の数文字だけを指定できます。連番ではなく、GitHub が発行する識別子でもありません。ローカルのコミットにも付きます。作成者が任意に付ける名前でもありません。名前を付ける仕組みはタグです。

ハッシュの計算には親コミットの ID も含まれるので、過去のコミットを書き換えると、それ以降のコミットの ID がすべて変わります。

ハッシュには、作成者、日時、メッセージなどのメタデータも入ります。2 人の開発者が、同じ親コミットの上に、まったく同じ内容のファイル変更をコミットしても、通常は異なる値になります。作業ディレクトリのパスや push した順序は、値に関係しません。内容と親が同じなら同じ値になる、という説明は誤りです。

```
git rev-parse --short=7 HEAD
```

```
c2e8783
```

ブランチとマージ [1.3]

ブランチは、独立した開発の流れです。機能の追加や修正を、他の作業に影響を与えずに進められます。Git のブランチは、コミットを指す軽量のポインターなので、作成も切り替えも高速です。変更を一時的に退避する保管領域ではなく、履歴を削除したものでもなく、リポジトリを別のアカウントへ複製したものでもありません。複製はフォークです。

ブランチの変更を取り込むことをマージと呼びます。feature ブランチの変更を main に取り込むには、取り込む側の main に切り替えて、取り込むブランチ名を指定します。

```
git switch main  
git merge feature
```

`git merge` は、現在チェックアウトしているブランチに、指定したブランチの変更を統合します。`git checkout feature` はブランチの切り替え、`git branch feature` はブランチの作成で、どちらもマージではありません。`git push`

`origin main` は手元の `main` をリモートに送る操作で、マージではありません。

同じファイルの同じ行が別々に変更されていると、マージでコンフリクト(競合)が起きます。競合した箇所は、次の3種類のマーカーで囲まれます。

```
<<<<<<< HEAD
two
=====
one
>>>>>>> x1
```

解消するには、人が内容を選んでファイルを編集し、競合マーカーを消します。解消したファイルを `git add` でステージして、コミットすると、マージが完了します。ブランチを削除して作り直しても、競合は消えません。マーカーを残したまま `push` して、リモートに解決させることもできません。新しい日時の変更が自動で優先されるわけでもありません。マージをやめたい場合は `git merge -abort` で元に戻せます。

clone / fetch / pull / push の違い [1.3]

リモートとのやりとりに使う4つのコマンドは、役割が違います。

コマンド	動作
<code>git clone <URL></code>	リモートリポジトリの履歴全体を、新しいディレクトリに複製する。初めて自分のPCに取得するときを使う
<code>git fetch</code>	リモートの変更を取得する。作業中のブランチには統合しない

コマンド	動作
<code>git pull</code>	<code>git fetch</code> に続けて、現在のブランチへ統合する。既定ではマージ
<code>git push</code>	手元のコミットをリモートへ送る

`git clone` では、既定のブランチがチェックアウトされた状態で作業を始められます。複製元は `origin` として登録されます。

`git fetch` は、取得するだけで作業ブランチには手を加えません。取得した内容は `origin/main` のようなリモート追跡ブランチに入ります。確認してから統合するかを選びたいときに使います。

```
git fetch origin
git log --oneline origin/main
```

```
98fe5ea c.txt を追加
62ce1fc a.txt を更新
4227b02 a.txt を追加
```

fetch のあとに `git status -sb` を実行すると、手元の main がリモートより遅れていることが分かります。

```
## main...origin/main [behind 1]
```

`git pull` を既定の設定で実行すると、リモートの変更を取得し、続けて現在のブランチへ統合します（設定によっては、マージの代わりにリベースで統合します）。pull は、手元のコミットをリモートへ送る操作ではありません。取得

だけして統合しない操作でもありません。リモート全体を新しいディレクトリに複製する操作は clone です。

ステージング(add → commit) [1.3]

Git では、変更をコミットする前に、次のコミットに含める変更を選びます。選んだ変更を置く場所を、ステージングエリアと呼びます。

```
作業ディレクトリ —git add—> ステージングエリア —git
commit—> ローカルリポジトリ
```

`git add` は、作業ディレクトリの変更をステージングエリアに登録します。新しいファイルを作る、変更をリモートへ送る、変更を履歴として確定する、のどれもありません。履歴として確定するのは `git commit` です。ステージングエリアがあるので、変更したファイルのうち一部だけをコミットに含められます。

a.txt と b.txt を変更して、a.txt の変更だけをコミットしたい場合は、次のようにします。

```
git add a.txt
git commit -m "a.txt を更新"
```

コミットに含まれるのは、ステージされた変更だけなので、`git add .` のあとでは b.txt も含まれます。`git push a.txt` という使い方はなく、`git commit` だけを実行すると、ステージされていない変更は含まれません。`git status` は、ステージ済みの変更と未ステージの変更を分けて表示します。

`git add` のあとで同じファイルを編集し、他に何もせず `git commit` を実行すると、`git add` した時点の内容がコミットされます。その後の編集は、含まれま

せん。コミットに入るのは、ステージされた時点のスナップショットです。そのあとの編集は未ステージの変更として残り、エラーにも取り消しにもなりません。 `git status -s` では、同じファイルが2列とも変更ありの `MM` が出ます。

```
MM a.txt
```

コミットのあとに中身を確認すると、入っているのはステージした時点の内容だけです。

```
a  
a2
```

作業ディレクトリのファイルは `a`、`a2`、`a3` の3行ですが、コミットされたのは `a3` を書き足す前の2行です。

コミットメッセージは、あとから履歴を読む人に変更の意図を伝えるもので、1行目に変更の内容と理由が分かる要約を書き、必要なら空行を挟んで詳細を書きます。日時と作者はメッセージに書く必要がありません。自動で記録されるからです。

HEAD と既定ブランチ [1.3]

`HEAD` は、いま作業している位置を示す参照です。通常は現在のブランチを指し、そのブランチの最新コミットを示します。リポジトリの最初のコミットでも、リモートの最新のコミットでも、最後に `push` したコミットでもありません。

```
cat .git/HEAD
```

特定のコミットを直接チェックアウトすると、`HEAD` がブランチを指さない状態になります。これを detached HEAD と呼びます。

既定ブランチは、リポジトリで最初に開かれるブランチです。次の 3 つの働きがあります。

- clone したときに最初にチェックアウトされる
- Pull Request の既定のマージ先になる
- リポジトリの設定で変更できる

最も古いブランチや、最後にマージしたブランチが自動で既定になるわけではありません。名前は `main` や `master` に固定されません。設定から別の名前に変えられます。新しく作るリポジトリの既定ブランチ名は、既定では `main` です。

個人アカウントと Organization の違い [1.4]

GitHub のアカウントには、個人アカウント、Organization、Enterprise アカウントの 3 種類があります。

種類	主な用途
個人アカウント	1 人の利用者がサインインして使う
Organization	複数人で共同管理する共有のアカウント
Enterprise アカウント	複数の Organization を一元管理する

Organization のアカウントでサインインすることはできません。メンバーは自分の個人アカウントでサインインします。

個人アカウントのリポジトリで開発してきた 5 人のチームが、メンバーの入れ替わりがあっても管理を続けたい場合は、Organization を作ってリポジトリを移し、チームとロールでアクセスを管理します。所有者が個人に依存せず、メンバーの追加や削除、チームごとの権限をまとめて管理できます。Organization には複数のオーナーを置けるので、特定の人があっても、残りのオーナーが管理を続けられます。各メンバーが自分のアカウントにフォークする運用、1 つのアカウントのサインイン情報を全員で共有する運用、リポジトリを public にして全員に書き込みを許す運用は、どれも管理の問題を解決しません。

個人アカウントが所有するリポジトリと比べて、Organization のリポジトリでは、チームを作成して、チーム単位でアクセス権を付与できます。private の設定、Issue と Pull Request の有効化、コラボレーターの招待は、個人アカウントのリポジトリでも使えるので、違いにはなりません。個人アカウントのリポジトリの権限は所有者と招待されたコラボレーターという単純な形で、Organization では、ロールやチームで細かく設定します。

Enterprise アカウントで束ねられるもの [1.4]

Enterprise アカウントは、複数の Organization をまとめて、ポリシー、請求、メンバーなどを一元的に管理するアカウントです。Enterprise の管理者というロールがあり、各 Organization の中のロールとは別に、全体の管理を担います。

会社の子会社ごとに 3 つの Organization を持っていて、請求とセキュリティのポリシーを会社全体で管理したい場合は、Enterprise アカウントを作成して 3 つの Organization を所属させます。3 つの Organization のオーナーを同じ 1 人に揃える方法や、3 つを 1 つに統合する機能、個人アカウントの Pro プランを契約して配下に追加する方法では、一元管理になりません。

Enterprise アカウントで管理できるのは、Organization やそのメンバー、ポリシーなどの設定です。複数の Organization にあるリポジトリのコードを同一の履歴にまとめる処理や、複数のリポジトリのブランチを 1 本に統合する処理は含まれません。個人アカウントの Free プランの機能の上限を引き上げる機能也没有ありません。リポジトリのコードの扱いは、各 Organization とリポジトリ側で決めます。

プラン(Free / Team / Enterprise)ごとの機能差 [1.4]

GitHub のプランは、Free、Pro (個人向け)、Team、Enterprise の順に機能が増えます。組織の private リポジトリで必要な機能の違いを、押さえておきます。

機能	必要なプラン
public リポジトリでのブランチ保護、コード所有者によるレビュー	Free でも使える
private リポジトリでのブランチ保護ルール、コード所有者によるレビューの必須化	Team 以上 (GitHub Team、GitHub Enterprise)
チームのレビュー依頼	Team 以上
internal リポジトリ、SAML シングルサインオン、Enterprise Managed Users	Enterprise (GitHub Enterprise Cloud)

private リポジトリでブランチ保護ルールとコード所有者のレビューを必須にしたい組織には、GitHub Team か GitHub Enterprise が必要です。GitHub Free のまま設定画面から有効にはできません。Marketplace のアプリを入れても、プランの制約は変わりません。Team にも用意されていて、Enterprise だけの機能でもありません。

GitHub Enterprise Cloud で使え、GitHub Team では使えない機能の代表が internal リポジトリです。Enterprise アカウント内の Organization 全体に公開する可視性で、Team で使えるのは private と public のリポジトリです。必須レビューを設定するブランチ保護ルール、private リポジトリ、チーム単位のアクセス権の付与は、Team でも使えます。SAML シングルサインオンと Enterprise Managed Users は、Team にはありません。

GitHub.com (Cloud) と self-hosted の選択 [1.4]

GitHub の提供形態は、次の 2 つです。

- GitHub.com (Cloud): GitHub が運用するサービスを使う形。組織ごとの機能を拡張する Enterprise 向けのクラウド型のプランが Enterprise Cloud
- GitHub Enterprise Server: 自社のインフラにインストールして運用するセルフホスト型の GitHub

Enterprise Server は、自社のサーバーやクラウドの環境に導入して、自分たちで運用します。アップグレードや保守も自社で行います。個人が PC にインストールして使うローカル専用の Git クライアントでも、リポジトリを閲覧するための読み取り専用のモバイルアプリでもありません。

法令上の理由で、ソースコードを自社のデータセンターの外に置けない企業は、自社のデータセンターに GitHub Enterprise Server を構築して運用します。GitHub.com 上の private や internal の設定は、アクセス制御で、データの保管場所を自社にする手段ではありません。Enterprise Cloud、GitHub Free の個人アカウントの private リポジトリ、GitHub Team の組織の private リポジトリは、どれも GitHub.com にコードを置きます。

GitHub Flow [1.5]

GitHub Flow は、軽量で、ブランチを中心にした協働の流れです。[^1]手順は次の 6 つです。

1. 既定ブランチから、作業用のブランチを作成する
2. ブランチで変更し、コミットして push する
3. Pull Request を作成する
4. レビューの指摘に対応する
5. Pull Request をマージする
6. 役目を終えたブランチを削除する

最初の手順はブランチの作成です。既定ブランチに直接コミットする運用でも、Pull Request を先に作る運用でもありません。Pull Request を作るには、ブランチに既定ブランチとの差分になるコミットが必要なので、ブランチ、コミット、Pull Request、レビューへの対応、マージの順になります。

ブランチ名は作業の内容が分かる短い名前にし、無関係な変更は別のブランチに分けます。

GitHub Flow の前提は、既定ブランチ (main) を常にデプロイできる状態に保つことです。そのため、main にはレビューを経た変更だけが入ります。変更のたびに直接コミットする運用、リリース直前にまとめて取り込む運用、開発中の不安定な変更を集める運用は、この前提と合いません。前提を守るために、ブランチ保護ルールで、既定ブランチへの直接の push を制限したり、Pull Request の承認を必須にしたりします。

動作確認をしていない変更を main へ直接 push して本番障害が起きたチームは、ブランチで作業し、Pull Request とレビューを経て main に取り込み、直接の push を制限する形に改めます。コミットメッセージの規則、月 1 回の

まとめ取り込み、週末にだけ push する運用では、確認していない変更が main に入る問題を防げません。

Git flow と比べると、GitHub Flow は、main と短命な作業ブランチを中心にした単純な流れです。develop や release を前提にしません。Git flow は、develop、release、hotfix などの長期のブランチを定めて、リリースの計画を重視するモデルです。GitHub Flow は手順が少なく、頻繁にデプロイする運用に向きます。モデルに優劣はなく、デプロイの頻度や、保守する版の数に合わせて選びます。複数の版を並行して保守し、リリースの計画を重視する運用なら、develop や release を定める Git flow も選べます。GitHub Flow は Git flow の上位版でも、ブランチ構成をすべて含む流れでもありません。リリースごとに専用のブランチを切ってマージを保留する流れも、GitHub Flow ではありません。

Markdown [1.6]

Markdown は、記号で書式を指定する軽い記法です。Issue、Pull Request、コメント、README で使います。

書きたいもの	記法
見出し	行頭に # を1つから6つ付け、半角スペースを入れる。# の数が多いほど小さい見出し
箇条書き	行頭に - とスペース
リンク	[表示する文字](URL)
太字	**太字** または __太字__
斜体	<i>*斜体*</i> または <i>_斜体_</i>

書きたいもの	記法
取り消し線	~~取り消し~~
引用	行頭に >
コードブロック	行頭の ``` で囲む
表	`

見出しは ****** で囲んでも、行頭に > や == を付けても、見出しになりません。
~~ は下線ではなく取り消し線で、* を 1 つずつ囲むと太字ではなく斜体になり、行頭の > はコードブロックではなく引用になります。

コメントでのメンションとして、@ にユーザー名を続けると、そのユーザーに通知が届きます。チームの場合は @組織名/チーム名 と書きます。# に番号を続けると、同じリポジトリの Issue や Pull Request へのリンクになります。ユーザー名の前に ! や # を付けても、通知は届きません。

タスクリストは、- [] で未完了、- [x] で完了です。チェックボックスとして表示され、クリックして切り替えることもでき、Issue や Pull Request の進捗に反映されます。

- [x] 設計を書く
 - [] テストを足す

コードブロックに言語を指定すると、構文が色分けされます。開始の ``` の直後に、python や json などの言語名を書きます。終了の側には書きません。バッククォートの数で言語は決まらず、言語名を省略すると色分けなしで表示されます。

```
```json
{"name": "octocat"}
```
```

長いログは、`<details>` と `<summary>` のタグで囲むと、最初は閉じておき、必要なときに開けます。`<summary>` に見出しの文字を書き、その後に空行を入れると、続く Markdown が正しく表示されます。

```
<details>
<summary>ログを表示</summary>

エラーの全文

</details>
```

コードレビューで、特定の行に具体的な修正案を示すには、コメントに `suggestion` のコードブロックで修正後の内容を書きます。コメントの入力欄にある提案のボタンを押すと、`suggestion` のブロックが自動で挿入されます。

```
```suggestion
print("hello")
```
```

`suggestion` を書くと、コメントに「提案を適用する」ボタンが表示されます。Pull Request の作成者が適用すると、Pull Request のブランチに、その内容を取り込む新しいコミットが追加されます。ベースのブランチに直接コミットされることも、提案がコメントの履歴だけに残ることも、承認済みの状態になる

こともありません。複数の提案を選んで、まとめて1つのコミットにすることもできます。言語を指定しないコードブロックで修正後の内容を貼っても、ボタンは付きません。

GitHub Desktop・Mobile・CLI・Web の使い分け [1.7]

GitHub を操作する手段は、複数あります。

| 手段 | 向く場面 |
|-------------------|---|
| GitHub Desktop | 画面の操作で、Git と GitHub の基本的な作業 (clone、ブランチの作成、コミット、push) を行うデスクトップアプリ。Windows と macOS 向け |
| GitHub Mobile | 外出先で、通知の確認、Issue と Pull Request の閲覧、コメント、レビュー (承認を含む) を行うアプリ。iOS と Android 向け |
| GitHub CLI (gh) | ターミナルから、Issue や Pull Request など GitHub の操作を行うコマンドラインツール |
| Web 画面 | ごく小さな修正を、ブラウザーだけで編集してコミットする |

GitHub Desktop は、コマンドの操作が苦手な人が、ローカルでブランチを作り、変更をコミットして GitHub に反映する場面に向きます。Desktop でコミットした内容は、「Push origin」の操作で GitHub に反映されます。GitHub Actions のジョブを実行するために PC を登録するエージェントでも、スマートフォンのモバイルアプリでも、ブラウザー上でコードを実行できるクラウド上の開発環境でもありません。

GitHub Mobile では、Git のコマンドの入力、リポジトリの clone とビルド、端末を GitHub Actions のランナーとして登録する操作はできません。外出中に緊急の Pull Request のレビュー依頼が届いたときは、GitHub Mobile で通知を開き、差分を確認して、承認のレビューを送ります。`git pull request approve` というコマンドはありません。GitHub Desktop は Pull Request の承認を送る手段ではありません。メール通知への返信は、Pull Request へのコメントとして投稿されるので、承認にはなりません。

GitHub CLI は、`gh pr create` や `gh issue list` のように、GitHub の機能を実行します。Git の代わりに履歴を管理するソフトでも、デスクトップアプリでも、読み取り専用のブラウザ拡張でもありません。コミットなどの Git の操作は、`git` コマンドで行います。

README の誤字を 1 か所だけ、手元に開発環境を用意せずに、手早く直したいときは、GitHub の Web 画面でファイルを開いて編集し、そのままコミットします。clone してビルド環境を整える方法や、`gh repo clone` を使う方法、Desktop で clone する方法は、1 か所の修正には手順が重く、向きません。Web 画面から、新しいブランチを作って、そのまま Pull Request を作成することもできます。

サンプルはここまでです

続き(第2章～巻末)は、GH-900 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

GH-900 GitHub Foundations 学習用テキスト(無料サンプル)

版

2026年10月版(GitHub Foundations (GH-900) 出題範囲 (Skills at a glance as of January 2026) 準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

GitHub、GitHub Copilot、GitHub Actions、GitHub Codespaces は GitHub, Inc. の商標または登録商標です。Microsoft は Microsoft Corporation の商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式の出題範囲 (Skills at a glance as of January 2026) と GitHub Docs の内容を基に作成したもので、GitHub, Inc. および Microsoft Corporation との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。